Este curso esta pensado para enseñar a programar en QuickBASIC 4.5 a quien así lo desee, si eres un programador de Pascal o C tal vez este curso te parezca aburrido. Tal vez tenga mentiras y/o errores, igual las ediciones anteriores, sin embargo mi intención es que les sea lo más útil posible. Quiero agradecer a www.lawebdelprogramador.com por poner a disposición de los internautas mi curso. Un cordial saludo y espero que lo disfruten.

Enero de 2009

Tadeo E. Ortega Ch.

Capítulo 0: Generalidades

Para crear un programa computacional es necesario conocer un lenguaje de programación, es decir, la herramienta que nos ayudará a mediar con los recursos de la computadora. A continuación vamos a hablar de algunos lenguajes de programación:

Lenguaje ensamblador
Este lenguaje nos permite crear programas para procesadores en particular.

Lenguajes de alto nivel
Al parecer los lenguajes de alto nivel se dividen en dos grupos: Compiladores e intérpretes. Al parecer, un **interprete** ejecuta las instrucciones del código fuente (el programa que nosotros escribimos) línea por línea; es decir nuestro programa no se ejecuta solo, sino que tiene que ser un programa reconocedor del lenguaje (intérprete) el que ejecute nuestro programa. Al parecer los **compiladores,** bajo MS-DOS, hacen lo siguiente: toman el código fuente, luego lo convierten en un código o programa objeto (**compilación**), que al parecer tiene una extensión .obj, para después combinarlo con las librerías (a este proceso se le llama **link** o enlace) y formar finalmente el programa en código reconocible por la máquina o programa ejecutable (con extensión .exe).

Nota: Existen dos conceptos fundamentales al hablar de programación: Programación estructurada y programación orientada a objetos (POO). Al parecer, la programación estructurada como su nombre lo dice, nos permite tener en un programa subprogramas que hacen cada uno una determinada tarea para el programa en general. Al parecer la POO es un tipo de programación en la que un programa esta formado por conjuntos de datos y subprogramas llamados "objetos".

Algunos de los lenguajes de alto nivel más difundidos son:

FORTRAN:

(**FOR**mula **TRAN**slator, *Traductor de fórmulas*) Al parecer es considerado como el primer lenguaje de alto nivel que se creó. Desarrollado allá por 1954, fue creado originalmente por IBM para ayudar a los ingenieros y científicos a trabajar con fórmulas matemáticas. Actualmente se enseña a estudiantes de ingeniería, y al parecer es utilizado por la NASA.

C:

(Llamado así por su predecesor, el lenguaje B) Al parecer fue desarrollado por Dennis Ritchie y Brian Kerninghan en los laboratorios AT&T en E.U, por ahí de 1974.

C++:

Es una versión orientada a objetos del lenguaje C, y al parecer fue creado en los 80's por Bjarne Strousptrup también en los laboratorios AT&T.

Java:

Al parecer fue creado a mediados de los 90's por la compañía Sun Microsystems basándose en el C++. Es un lenguaje de POO que corre en una "maquina virtual", de la cual hay versiones para diferentes sistemas operativos. Por eso el lema de Java es "*Write once, run everywhere*" ("Escribe una vez, corre dondequiera").

Pascal:

(Al parecer, en honor al matemático francés Blaise **Pascal**) Al parecer fue creado por el profesor suizo Niklaus Wirth con el propósito de enseñar programación, allá por 1973. Aunque existen varias versiones del Pascal como TMT Pascal y Microsoft QuickPascal, al parecer el Pascal ganó popularidad gracias a Turbo Pascal, que fue un compilador lanzado al mercado por la compañía Borland International a principios de los 80's. Al parecer Turbo Pascal era un compilador que se lanzó al mercado como un experimento, pero resulto ser todo un éxito ya que era veloz, valía al parecer $ 49.99 (US) y trabajaba en sistemas IBM. Al parecer, posteriormente el Turbo Pascal se fue actualizando hasta llegar a la versión 7 que soportaba POO desde la versión 5.5.

Breve historia del BASIC

BASIC: (Acrónimo de **B**egginers **A**ll-purpouse **S**imbolic **I**nstruction **C**ode, *Código de instrucciones simbólicas multiuso para principiantes*) Al parecer, fue desarrollado por Tomas Kurtz y John Kemeny en Dartmouth College como un dialecto del Fortran para enseñar programación de una manera fácil a los que en ese tiempo (por ahí de 1964) se querían introducir al mundo de la informática. El BASIC es un lenguaje facil y es muy popular. Al parecer cuando recién se crearon las PC, el lenguaje BASIC se convirtió en la primera lengua franca de las computadoras gracias a Microsoft BASIC, ya que este venía instalado junto con los ordenadores IBM y su uso era muy fácil. Se cree que ganó su popularidad debido a algunas implementaciones como Apple BASIC (para computadoras Apple), y sobre todo a GW-BASIC y BASICA (ambos de Microsoft). Por ese entonces, vinieron dos potentes versiones del BASIC: Microsoft QuickBASIC y Turbo Basic (de Borland). Ambas versiones presentaban un lenguaje estructurado, compilado, considerablemente mejorado y muy amigable con el programador tanto novato como avanzado. Aunque se dice que se trató inútilmente de crear un estándar, este lo vinieron a marcar los BASIC de Microsoft. QuickBASIC desde la versión 4.0 (hasta donde sé) contaba con un completo sistema de ayuda que mejoró sustancialmente en la versión 4.5, soporte de ratón; capacidad de obtener amplia ayuda, ejemplos y pormenores sobre una sentencia con solo dar un *clic* con el botón secundario del ratón sobre ella, lo cual resultaba muy útil, cómodo y amigable. Además contenía otras muchas más comodidades con respecto al Turbo como tipos de variables definidos por el usuario, editor inteligente, un gestor de ventanas más potente, y capacidad de declarar variables y constantes de una manera más potente. Posteriormente se fue perfeccionando hasta llegar a su versión 6.0, al parecer llamada desde ahí y hasta la versión 7.1 "*BASIC PDS*" (**P**rofesional **D**evelopment **S**ystem, *Sistema de desarrollo profesional*). Al parecer, posteriormente Microsoft distribuyó junto con las últimas versiones de MS-DOS un interprete llamado Qbasic, del cual salió la versión 1.0 y luego la 1.1 que corregía algunas deficiencias de su predecesor.

Mas o menos principios de los 90's, Microsoft desarrolló Visual Basic para Windows y Visual Basic para MS-DOS, que permitian crear aplicaciones para Windows y DOS respectivamente.

Actualmente existe Visual Basic .NET, que es al parecer una version orientada a objetos del lenguaje BASIC.

Capítulo 1: Entrada y salida básica

SENTENCIAS REM, PRINT, CLS, END

Escribamos en el editor de QB el siguiente programa:

```
rem Simple saludo (saludo.bas)
rem 17/08/2000
cls
print "¡¡¡ BIENVENIDO A QUICKBASIC 4.5 :-) !!!"
end
```

¿Qué es lo que hace el programa anterior?. Presiona <F5>. Nos muestra en la pantalla el texto *"¡¡¡ BIENVENIDO A QUICKBASIC 4.5 :-) !!!"*. Ahora, lo que hace la sentencia **REM** (abreviatura de la palabra inglesa *remark*) es insertar un comentario dentro de nuestro programa fuente; esto es, el comentario no será tomado en cuenta por el compilador, solo sirve para insertar datos que sirvan como referencia al programa, para saber quien lo hizo, cuándo, para qué sirve, etc. Es bueno insertar comentarios de manera regular en nuestros programas como referencia a una orden o a algo que sea de interés para nosotros o para quien vaya a utilizar nuestro programa, especialmente si trabajamos en grupo o distribuimos nuestras fuentes. Lo que sigue es la sentencia **CLS** que borra la pantalla. Es primordial limpiar la pantalla cada vez que iniciemos un programa, esto para poder trabajar mejor y que se vea mejor. Para los que todavía no captan, **PRINT** es la orden que nos permite la salida por la pantalla, en este caso es la cadena de texto *¡¡¡ BIENVENIDO A QUICKBASIC 4.5 ;-) !!!*. Cuando queremos que con la orden PRINT salga texto, este ha de ir entre comillas dobles ("").

Para los que no estén familiarizados con los emoticones, estos son formas de representar expresiones por medio de caracteres; son utilizados en el chat para ahorrar tiempo y escribir menos. Por ejemplo el emoticón ;**-)** de lado parece una cara guiñando un ojo, ¿ya?. Cada quien puede crear sus emoticones según tenga imaginación, algunos son :

:-)	Carita sonriente
<|:-)	De fiesta
0:-)	Niño bueno
;-)	Guiñando un ojo
:.-(	Llorando
:-(	Tristeza
(.) (.)	Busto
:-0	Asombro
d:-)	Cachucha de lado
:^	Un respiro
]:->	Diablillo
:-x	Me voy a callar
:-I	Serio
--←←@	Una flor
:-{	Alguien con bigote

La sentencia **END** nos indica que nuestro programa ha terminado.

SENTENCIA COLOR

Ahora analizemos este otro programa:

```
REM Salida de texto con color (saludo2.bas)

CLS
COLOR 10        'Color verde brillante
PRINT "¡¡¡ BIENVENIDO A QUICKBASIC 4.5 :)!!!"
END
```

En realidad este programa hace lo mismo que el anterior, solo que el texto sale con un color verde. La sentencia **COLOR** nos permite dar color al texto que saldrá por la pantalla:

Sintaxis:

COLOR principal [,de fondo] [,de borde]

- El color principal va del 0 al 15 y del 16 al 31 con parpadeo.
- El color de fondo es opcional y va del 0 al 7
- El color de borde es opcional y va del 0 al 15

Pero ¿qué es ese ' y el texto que esta enseguida?. Como se puede adivinar, el apóstrofe ('') es otra manera de insertar un comentario y es lo mismo que REM.

Si sales a DOS cuando termines de ejecutar el programa, verás que el prompt quedará del color del texto (verde). Para que el símbolo de DOS quede con su color original, solo hay que agregar COLOR 7 (Gris) o CLS antes de END, solo que si le aplicamos el CLS es obvio que el texto que imprimimos se borrará inmediatamente antes de terminar el programa. Luego veremos como mejorarlo.

CARACTERES Y OPERACIONES BÁSICAS

Veamos ahora como realizar las operaciones más básicas:

```
REM Manejo de operadores (operador.bas)

numero1 = 5
numero2 = 5
suma = numero1 + numero2
multi = numero1 * numero2
divi = numero1 / numero2
resta = numero1 - numero2
expo = numero1 ^ numero2

CLS
PRINT "La suma de"; numero1; "más"; numero2; "es igual a"; suma
PRINT "El producto de"; numero1; "por"; numero2; "es igual a"; multi
```

```
PRINT "El cociente de"; numero1; "entre"; numero2; "es igual a"; divi
PRINT "La resta de"; numero1; "menos"; numero2; "es igual a"; resta
PRINT “El resultado de elevar”; numero1;”a la”; numero2 ;”es”; expo
END
```

Como se puede apreciar en el ejemplo anterior, hacemos 5 operaciones básicas con 2 números ambos iguales a 5 y guardamos el resultado en cuatro **variables** cuyo nombre indica más o menos que operación se hizo con los números. Luego imprimimos en la pantalla 4 mensajes alusivos a las diversas operaciones que realizamos utilizando las variables *numero1* y *numero2*. Cuando imprimimos variables, **estas no van entre comillas** ya que si las ponemos entre comillas QB las entenderá como si fueran texto e imprimirá solo el nombre de la variable. Por esto pusimos punto y coma (**;**) para diferenciarlas del texto y además indicar que después del texto se deje un espacio para que no se amontone el número con el texto anterior.

Veamos este otro ejemplo:

```
REM Manejo PRINT y variables (prin1var.bas)

numero1 = 5
numero2 = 27

CLS
PRINT "El primer número es igual a"; numero1
PRINT "El segundo número es igual a"; numero2
PRINT "La suma de ambos es igual a"; numero1 + numero2
END
```

Como podemos ver, **también podemos realizar operaciones dentro** de la misma sentencia PRINT y no hay problema. Las operaciones básicas son las siguientes:

Operador	Función
*	Multiplicación
+	Suma
-	Resta
^	Exponenciación
/	División

Existe también el operador **MOD** que da el residuo de una división de dos números enteros hasta donde el cociente es entero. Por ejemplo 14 MOD 5 retorna 4:

$$\begin{array}{r} 2 \\ 5\overline{)14} \\ \hline 4 \end{array}$$

Hasta aquí el cociente es entero y el **residuo** es 4.

Si por ejemplo ponemos 14 MOD 4.6, entonces 4.6 será **redondeado** a 5 ya que no podemos utilizar reales con MOD, solo enteros.

También existe el operador *división entera* que se representa por la diagonal invertida (****) y cuya función es obtener el cociente entero de una división. Siguiendo con el ejemplo anterior, 14 \ 5 retornara 2:

$$\begin{array}{r} 2 \\ 5\overline{)14} \\ \hline 4 \end{array}$$

Hasta aquí el **cociente** es entero.

Ahora analizemos el siguiente ejemplo:

```
REM Manejo de operadores (triangu.bas)

baze = 10        'Debe ser baze y no base. Ojo...
altura = 5
CLS
PRINT "El área de un triángulo cuya base mide"; baze;
PRINT "y cuya altura mide"; altura; "es:"
PRINT (baze * altura) / 2
END
```

Como podemos apreciar, el punto y coma **al final** de una sentencia PRINT sirve para juntar en una misma línea de la pantalla otra sentencia PRINT que esta en otro renglón en nuestra fuente. ¿Si ponemos una coma (,) en vez de punto y coma que pasará?. Ponle (,) en vez de (;) y córrelo (<F5>). Como puedes ver, lo que esta después de una coma se separa una tabulación (como 7 espacios).

EL MODIFICADOR TAB

El modificador **TAB**(n) va después de PRINT y nos permite colocar texto en una determinada posición del renglón en uso, y donde n es la posición donde queremos que se empiece a imprimir nuestro texto:

```
REM Uso de TAB (tab.bas)

CLS
PRINT TAB(3); "Hello, world!"
PRINT TAB(23); "¡Hola, mundo!"
END
```

Aquí el texto *Hello, world!* se imprimirá después de 3 espacios, y el texto *¡Hola mundo!* en otro renglón y desde la posición 23.

Nota: La pantalla en modo texto consta de 70 columnas y 25 renglones.

TIPOS DE DATOS

Los tipos de datos que se manejan en QuickBASIC son los siguientes:

Nombre	Descripción	Rango	Declaración	Espacio en memoria
Integer	Entero corto con signo	-32768 a 32767	%	2 bytes (16 bits)
Long	Entero largo con signo	-2,147,483,648 a 2,147,483,647	&	4 bytes (32 bits)
Single	Real de simple precisión con signo y con 7 decimales	1.5 E-45 a 3.4 E38	!	4 bytes (32 bits)
Double	Real de doble precisión con signo y 15 decimales	5.0 D-324 a 1.7 D308	#	8 bytes (64 bits)
String	Cadena de caracteres	0 a 256 caracteres	$	1 byte p/caracter

Todas las variables que utilizamos sin declarar el tipo se asumen como **Single**. Pero...¿Cómo declaramos los tipos de las variables?. Para declarar, por ejemplo, una variable que vaya a almacenar una dirección de una persona (por lo tanto será de tipo **String**) que se llame *direccion* la declaramos de la siguiente forma:

direccion**$** = "Morelos y 5ª S/N, Col. Portales"

Para declarar una variable de un tipo basta con **poner el identificador al final.** Ahora, cuando nosotros guardamos una cadena de texto en una variable de tipo String, el texto debe de ir entre comillas dobles ("").

Para declarar una variable de tipo real, por ejemplo de simple precisión (Single) y que vaya a guardar el numero 5.123×10^{24} la declaramos de la siguiente forma:

numero**!** = 5.123**E**24

Ahora una de doble precisión (Double) que se llame numero2 y que tenga el valor de 1.45877×10^{-298}

numero2**#** = 1.45877**D**-298

Observación: Para declarar un exponencial en una variable Single utilizamos **E** y para declararlo en una variable de tipo Double utilizamos **D**.

```
'--------------
'Ejemplo de tipos de datos (tipos.bas)
'--------------

cadena$ = "Soy una cadena  "
enterocorto% = 1200
enterolargo& = 198108
simplereal! = 1.5688
doblereal# = 3.458D+301

CLS
```

```
PRINT cadena$
PRINT "Yo soy un entero corto:"; enterocorto%
PRINT "Yo soy un entero largo:"; enterolargo&
PRINT "Yo soy un real simple:"; simplereal!
PRINT "Yo soy un real doble:"; doblereal#
END
```

RESTRICCIONES Y NOTAS IMPORTANTES

Una de las ventajas y a la vez desventaja que tiene el lenguaje BASIC frente a otros lenguajes de programación es que podemos crear variables en cualquier parte del programa según las vayamos necesitando. Cuando creamos una nueva variable, QB automáticamente reserva un espacio en memoria para almacenar el valor de esa variable, en el caso de los números la variable almacena un 0 y en el caso de las cadenas se almacena la cadena nula ("""). Esta es un arma de dos filos ya que podemos caer en errores como el siguiente:

```
numeroquevalepi=3.141592

CLS
PRINT "El valor de pi es"; numeroqevalepi
END
```

Nosotros **declaramos** una variable (Single por default) que se llama *numeroquevalepi* y que almacena el valor 3.141597 y al tratar de imprimirla **escribimos mal** el nombre de la variable y ponemos *numeroqevalepi* (sin la **u**) y lo que hicimos fue **crear una nueva** variable que por default **toma el valor de 0 en memoria**, así lo que se imprimirá en pantalla será:

El valor de pi es 0

Esto en programas cortos es fácil de detectar, pero en programas largos es un poco más difícil; aun así, esto permite una programación más fluida, y es ideal para usar variables "sobre la marcha", cosa que no tienen otros lenguajes como Pascal o Modula-2. Con la práctica es muy fácil de detectar este tipo de errores. Por fortuna Microsoft, pensó en esto y agregó la opción de requerir la declaración de las variables en el Visual Basic; o sea, si una variable es hallada en el programa y no tiene declaración, se marca un error.

Otra observación es que **no podemos usar palabras reservadas** para nombrar una variable, por ejemplo, no podemos utilizar una variable que se llame Print, Base, Rem, Cls, End, Len, Right$, Color, etc. ni en mayúsculas ni en minúsculas; tambien las variables **no pueden empezar por un número o por un identificador** de tipo como #perro, !cosa, 3cadenas, etc., **ni** tener **espacios**, **operadores**, la **ñ** o el **guión bajo** (_).

Una manera más fácil de declarar el tipo de una variable es con **DIM**.

ORDEN DIM

Permite la declaración de variables de un tipo determinado.

```
'--------------
'Ejemplo de tipos de datos usando DIM (tipos2.bas)
'--------------

DIM cadena AS STRING
DIM enterocorto AS INTEGER
DIM enterolargo AS LONG
DIM simplereal AS SINGLE
DIM doblereal AS DOUBLE

cadena = "Soy una cadena  "
enterocorto = 1200
enterolargo = 198108
simplereal = 1.5688
doblereal = 3.458D+301

CLS
PRINT cadena$
PRINT "Yo soy un entero corto:"; enterocorto
PRINT "Yo soy un entero largo:"; enterolargo
PRINT "Yo soy un real simple:"; simplereal
PRINT "Yo soy un real doble:"; doblereal
END
```

La orden DIM permite una mejor manera de declarar variables y evita tener que poner los identificadores de tipo cada vez que la utilicemos, lo cual ahorra una muy tediosa tarea en programas largos.

Nota: Si estas utilizando Turbo Basic (de Borland) no puedes declarar variables de esta manera.

ENTRADA DE DATOS: LA ORDEN INPUT

Hasta ahora hemos visto como guardar valores y como sacarlos por la pantalla, ahora vamos a dejar que el usuario los introduzca:

```
' Ejemplo de INPUT (input.bas)

CONST pi = 3.1416  ' Con CONST se declara una constante cuyo valor no
                   '  cambiará durante el programa (solo QB).

CLS : COLOR 10     ' Podemos utilizar mas de dos ordenes en una misma línea
                   ' separándolas por dos puntos (:).

PRINT TAB(33); "AREA DE UN CIRCULO"
PRINT : PRINT
COLOR 7
```

```
INPUT "¿Cuál es el radio del círculo"; radio
area = pi * (radio ^ 2)
PRINT "El área del circulo es"; area
END
```

Pudimos ver en el programa anterior, en la sentencia **INPUT**:

- Un letrero (opcional) entre comillas, seguido de un (;) que le pondrá un signo de interrogación al final del letrero al salir por la pantalla.
- Una variable donde se almacenara un valor.

En el caso de que queramos que el usuario nos dé 2 o más mas valores en una misma sentencia INPUT se hace de la siguiente manera:

```
' Ejemplo de INPUT con mas de un valor (input2.bas)

CLS : COLOR 10
PRINT TAB(33); "AREA DE UN TRIANGULO"
PRINT : PRINT
COLOR 7
INPUT "Introduzca la base y la altura del triángulo (separados entre sí por una coma) => “, b, h
a = (b * h) / 2
PRINT "El área del triángulo es"; a : END
```

Si no queremos que aparezca un signo de interrogación (?) al final de nuestro letrero debemos ponerle una coma (,) . Para almacenar más de una variable en una misma sentencia INPUT, estas deben ir **separadas entre si** por comas; de igual manera el usuario debe de introducir los datos que se le piden separados entre si por una coma. Por ejemplo, si el programa anterior pide

Introduzca la base y la altura del triángulo (separados entre sí por una coma) => _

El usuario puede introducir

5.6, 4

Capítulo 2: Sentencias de control selectivas

OPERADORES RELACIONALES

Los **operadores relacionales** nos sirven para determinar la **relación** que tiene una expresión con otra. Los operadores relacionales que manejaremos en QB son los siguientes:

Operador o signo	Significa	Ejemplo	Se lee
=	Igual que	x = y	x igual a y
<	Menor que	x < y	x menor que y
>	Mayor que	x > y	x mayor que y
<=	Menor o igual	x <= y	x menor o igual a y
>=	Mayor o igual	x >= y	x mayor o igual a y
<>	Diferente de	x <> y	x diferente de y

SENTENCIA IF...THEN...ELSE

Muchas veces se nos presentan situaciones en las que tenemos que evaluar una condición o situación. **Si** la condición es **verdadera** o "cumple", **entonces** ejecutamos una o más instrucciones; **si no**, ejecutamos otra u otras instrucciones. Veamos el siguiente ejemplo:

```
CLS
PRINT TAB(20); "VALIDA SI UN NUMERO ES NEGATIVO O POSITIVO"
PRINT : PRINT
INPUT "Introduce un entero -> ", numero

IF numero < 0 THEN
   PRINT "El número es negativo"
ELSE
   PRINT "El número es positivo o cero"
END IF
END
```

El programa anterior valida si un entero es positivo o cero, o negativo, de la siguiente forma:

- El programa pide un número.
- **Si** (if) el numero introducido es menor que cero (esta es la condición a evaluar) **entonces** (then) se imprime el mensaje "*El número es negativo*".
- **Si no** (else) es negativo, entonces se imprime el mensaje "*El número es positivo o cero*".

Un bloque de instrucciones **IF...THEN...ELSE** debe terminar siempre con **END IF** para indicar que nuestro proceso de selección IF ha terminado. Ahora, las condiciones que se manejan en las sentencias IF pueden ser **verdaderas** (True, en inglés) o **falsas** (False). Si por ejemplo corremos el programa anterior y al pedirse el numero nosotros introducimos un –1, entonces la sentencia IF verá que la condición es verdadera (en efecto el número es menor que 0) y ejecutará la sentencia o sentencias que siguen al IF hasta encontrar el ELSE, luego se pasará hasta el END IF para luego seguir con el curso del programa. De otra forma si nosotros introducimos un 81, entonces la sentencia IF verá que las condición es falsa (false) y no ejecutará las sentencias que le siguen, luego se pasará hasta la sentencia ELSE y se ejecutaran las sentencias que le siguen hasta que termine el bloque (END IF), para luego continuar el curso del programa.

Las condiciones suelen también ser afectadas por los **operadores lógicos**. Los operadores lógicos nos sirven para evaluar condiciones tales como:

- "Si numero1 no es igual a cero entonces..."
  (IF **NOT** numero1 = 0 THEN...)
- "Si numero1 es igual a cero o mayor a 1 entonces..."
  (IF numero1 = 0 **OR** numero1 > 1 THEN...)
- "Si numero1 es igual a 1 y numero2 es menor que cero entonces..."
  (IF numero1 = 1 **AND** numero2 < 0 THEN)
- "Si numero1 es igual a cero ó menor que 0 entonces..."
  (IF numero1 = 0 **XOR** numero < 0 THEN)

Valores de:		**Valor retornado por el operador lógico:**					
x	**y**	**NOT x**	**x AND y**	**x OR y**	**x XOR y**	**x EQV y**	**x IMP y**
V	V	F	V	V	F	V	V
V	F	F	F	V	V	F	F
F	V	V	F	V	V	F	V
F	F	V	F	F	F	V	V

Para que esto quede más claro, veamos lo siguiente que nos ayudarán a ver más claro que son los operadores lógicos:

NOT

Una chica le pregunta a otra:

- *"¿Vas a ir al baile?.* La otra contesta:
- *No, no iré.*
- *Si no vas al baile no verás a Fernando Antonio Del Valle Santiesteban...*

Aquí si la chica no va al baile, entonces no podrá ver a Fernando A. Del V. Santiesteban. Si va, entonces lo verá.

```
INPUT "¿Irá Petra al baile (si = 1, no = 0)?. Introduzca un número ", baile
si = 1
no = 0

IF NOT baile = si THEN
   PRINT "No verá a Fernando"
ELSE
    ? "Verá a Fernando"            ' El signo ? es lo mismo que PRINT
END IF
END
```

AND

Una persona va a sacar la cartilla y la secretaría le dice:

- *Necesitas el acta de nacimiento original y un comprobante de domicilio.*

Para que esa persona pueda obtener su cartilla necesita el acta **y** un comprobante de domicilio. Si le falta **cualquiera** de estos papeles entonces no se la dan.

```
PRINT "Introduzca la respuesta siendo SI = 1 Y NO = 0
INPUT "¿Trajo el acta de nacimiento -> "; acta
INPUT "¿Trajo el comprobante de domicilio -> "; domicilio
si = 1
no = 0

IF acta = si AND domicilio = si  THEN
   PRINT "Toma tu cartilla"
ELSE
   PRINT "Vaya por lo que le falta y vuelva cuando tenga todos los papeles"
END IF
END
```

OR

El niño dice a su mamá en la nevería:

- *¿Mamá me compras una nieve?.* La mamá le dice
- *De cual quieres hijo, de fresa o de limón.* El niño le contesta:
- *Quiero de las dos.*
- *Esta bien, hijo.*

Aquí si el niño escoge de fresa, de limón o de ambas, de todas maneras comerá nieve.

```
INPUT "¿Quieres nieve de fresa (si = 1, no = 0)"; fresa
INPUT "¿Quieres  nieve de limón (si = 1, no = 0)"; limon
si = 1
no = 0

IF fresa = si OR limon = si THEN
    PRINT "Toma tu nieve"
ELSE
    PRINT "No comerás nieve"      ' Si no quisiera ninguna de las dos
END IF
END
```

A este tipo de o se le llama **o inclusivo**, ya que puede **incluir** a ambos.

XOR

El niño le dice a la mamá en la nevería:

- *¿Mamá me compras una nieve?.* La mamá le dice:
- *De cual quieres hijo, de fresa ó de limón.* El niño le contesta:
- *Quiero de las dos.*
- *No, hijo: fresa ó limón, de las dos no.*

```
? "Escoge solo un sabor (si = 1, no = 0)
INPUT "¿Quieres nieve de fresa"; fresa
INPUT "¿Quieres nieve de limón"; limon
```

```
si = 1
no = 0

IF fresa = si XOR limón = si THEN
‘ Si fresa y limón son si,  la condición no se cumple (XOR retorna false) y por lo tanto no se ejecuta.
     ? “Toma tu nieve”    ‘ Si el niño quiere solo alguna de las dos
ELSE
     ? “Fresa o limón, pero no de las dos
END IF
END
```

A este ó le llamamos también **ó exclusivo** porque **excluye** a una opción.

EQV

Cuando soltamos un objeto desde cierta altura este experimenta en su caida una aceleracion en su velocidad.

```
CLS
INPUT “¿Soltaste el objeto (si = 1, no = 0)”; soltar
INPUT “¿Aumento su velocidad (si = 1, no =0)”;delta
si = 1
no = 0

IF soltar = si EQV delta = si THEN
   PRINT “Correcto :-)”
ELSE
   PRINT “Mentiroso ;-)”
END IF
END
```

IMP

“Si estudias para el examen pasarás”. Aquí estamos afirmando que si se estudia, entonces se pasará.

```
CLS
? “si = 1, no = 0”
INPUT “¿Estudiaste ->”; estudiar
INPUT “¿Pasaste el examen ->”; examen
si = 1
no = 0

IF estudiar = si IMP examen = si THEN
   ? “Si te creo”
ELSE
   ? “Tal vez”
END IF
END
```

Nota: Puede que alguien piense en cosas como que pasaría si tuviéramos algo como

a = x AND y

en un programa. En este caso los operadores lógicos producen **operaciones binarias** entre los valores de las variables.

IF ANIDADOS

Dentro de un programa podemos evaluar ciertas condiciones utilizando **sentencias IF dentro de** más bloques IF...THEN...ELSE. A este conjunto de sentencias agrupadas se les llama "**IF anidados**".

```
CLS
INPUT "Introduce un número del 1 al 10-> “, n
PRINT "El número introducido fue ";
COLOR 10

IF n < 5 THEN
  IF n = 1 THEN PRINT "uno"
  IF n = 2 THEN PRINT "dos"
  IF n = 3 THEN PRINT "tres"
  IF n = 4 THEN PRINT "cuatro"
ELSE
  IF n = 5 THEN PRINT "cinco"
  IF n = 6 THEN PRINT "seis"
  IF n = 7 THEN PRINT "siete"
  IF n = 8 THEN PRINT "ocho"
  IF n = 9 THEN PRINT "nueve"
  IF n = 10 THEN PRINT "diez"
END IF
COLOR 7
END
```

El programa anterior simplemente divide los posibles valores de n en dos partes: los números menores que 5, y los números <u>iguales o menores</u> que 5. En el ejemplo podemos ver algunas cosas:

- La sentencias IF pueden ejecutarse **sin necesidad de que haya un ELSE** después. O sea, si la sentencia se cumple la orden se ejecuta y si no, no.
- Si la sentencia IF solo tiene **una orden** que ejecutar, esta puede ir inmediatamente después del THEN.
- El END IF solo se pone cuando existe un bloque IF...THEN..ELSE, o cuando existe un IF que si se cumple ejecutará varias instrucciones, sin que después haya un ELSE; esto es:

  IF condición THEN
    instrucción **1**
    instrucción **2**
    instrucción **3**
    ...
    instrucción **n**
  END IF

Ahora veamos un ejemplo algo más complejo:

```
CLS
PRINT "PROGRAMA QUE DETERMINA SI UN CARACTER ES:"
PRINT "1) Letra minúscula de 'a' hasta 'e' "
PRINT "2) Letra mayúscula de 'A' hasta 'E' "
PRINT "3) Numero del 1 al 5"
PRINT "4) Operador  matemático"
PRINT "5) Otro caracter"

PRINT
COLOR 7
INPUT "Introduce un caracter -> ", c$
PRINT "El caracter introducido es ";
COLOR 10

IF c$ = "1" XOR c$ = "2" XOR c$ = "3" XOR c$ = "4" XOR c$ = "5" THEN
  PRINT "número"
ELSE
  IF c$ = "+" XOR c$ = "-" XOR c$ = "*" XOR c$ = "/" XOR c$ = "\" XOR c$ = "^" THEN
    PRINT "operador  matemático"
  ELSE
    IF c$ = "a" XOR c$ = "b" XOR c$ = "c" XOR c$ = "d" XOR c$ = "e" THEN
      PRINT "letra minúscula"
    ELSE
      IF c$ = "A" XOR c$ = "B" XOR c$ = "C" XOR c$ = "D" XOR c$ = "E" THEN
        PRINT "letra mayúscula"
      ELSE
        PRINT "otro caracter diferente de letra (a – e), número (1 – 5) u operador matemático"
      END IF
    END IF
  END IF
END IF
END
```

- Como el caracter introducido solo puede ser uno particular usamos XOR.
- Existen bloques IF...THEN...ELSE anidados.
- Si un IF no se cumple, entonces se salta al ELSE inmediato que da lugar a otro IF.
- Cada bloque IF...THEN...ELSE termina con el END IF que le queda más cerca. Por ejemplo, el ultimo bloque IF...THEN...ELSE termina con el primer END IF; luego, el bloque que contenía a este bloque, termina con el segundo END IF, y así sucesivamente.

ELSEIF

En pocas palabras, **ELSEIF** es un "*IF inmediatamente después de un ELSE*". Veamos el ejemplo anterior usando ELSEIF.

```
INPUT "Introduce un caracter -> ", c$
PRINT "El caracter introducido es ";
COLOR 10

IF c$ = "1" XOR c$ = "2" XOR c$ = "3" XOR c$ = "4" XOR c$ = "5" THEN
  PRINT "número"
ELSEIF c$ = "+" XOR c$ = "-" XOR c$ = "*" XOR c$ = "/" XOR c$ = "\" XOR c$ = "^" THEN
```

```
   PRINT "operador  matemático"
ELSEIF c$ = "a" XOR c$ = "b" XOR c$ = "c" XOR c$ = "d" XOR c$ = "e" THEN
   PRINT "letra minúscula"
ELSEIF c$ = "A" XOR c$ = "B" XOR c$ = "C" XOR c$ = "D" XOR c$ = "D" THEN
   PRINT "letra mayúscula"
ELSE
   PRINT "otro caracter diferente de letra (a – e), número (1 –  5) u operador matemático"
END IF
END
```

- La sentencia ELSEIF es también una parte opcional de un bloque IF...THEN...ELSE. Podemos también tener uno o varios ELSEIF dentro de un solo bloque IF...THEN...ELSE
- Si el primer IF no se cumple, QB automáticamente va hasta el ELSEIF donde la condición se cumple y evalúa las ordenes que están después de este, hasta que se topa con otro ELSEIF, entonces saldrá del bloque IF...THEN...ELSE.
- Si ni el IF ni los ELSEIF se cumplen, entonces el programa se va hasta ELSE.

ERRORES FRECUENTES Y CONSIDERACIONES

Como ya vimos anteriormente, podemos tener programas como el siguiente en el que haya IF "sueltos", es decir, sin ELSE y sin END IF.

```
CLS
INPUT "Un número (1 – 4) -> ", n
IF n < 3 THEN
   IF n = 1 PRINT "uno"
   IF n = 2 PRINT "dos"
ELSE
   IF n = 3 PRINT "tres"
   IF n = 4 PRINT "cuatro"
END IF
END
```

Sin embargo, **no debemos** poner ELSE "sueltos" (o *colgantes*), es decir, sentencias ELSE que vayan después de otras sentencias ELSE y que además **no** contengan ninguna condición a evaluar, es decir, que no contengan un IF después de ellas:

```
CLS
INPUT "Número (1-3) -> ", n
IF n = 1 THEN
   PRINT "uno"
ELSE
   PRINT "dos"
ELSE            ' No hay condición a evaluar, solo una orden PRINT enseguida.
   PRINT "tres"
END IF
END
```

Lo que hará el programa anterior es evaluar el IF, si se cumple se saltará el primer ELSE **pero no el segundo** y lo ejecutará. Luego si hubiera más ELSE "sueltos", ejecutará uno no y otro sí, uno no y otro sí, así sucesivamente. En el caso de que la condición no se cumpliera, se ejecutará

el primer ELSE y se saltará el siguiente hasta terminar el bloque. Si hubiera mas ELSE se saltaría el siguiente ELSE y ejecutaría el otro, se salta uno y ejecuta el otro, y así sucesivamente. Lo correcto podría ser:

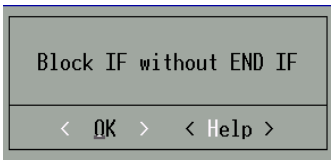
```
CLS
INPUT "Número (1-3) -> ", n

IF n = 1 THEN
   PRINT "uno"
ELSEIF n = 2 THEN
   PRINT "dos"
ELSEIF n = 3 THEN
   PRINT "tres"
ELSE
   PRINT “fuera de rango”
END IF
END
```

Un error que se comete muy frecuentemente cuando se anidan IF es que nos falte poner un END IF. Ejemplo:

```
CLS
PRINT "PROGRAMA QUE DETERMINA SI UN CARACTER ES:"
PRINT "1) Letra minúscula de 'a' hasta 'e' "
PRINT "2) Letra mayúscula de 'A' hasta 'E' "
PRINT "3) Numero del 1 al 5"
PRINT "4) Operador  matemático"
PRINT "5) Otro caracter"

PRINT
COLOR 7
INPUT "Introduce un caracter -> ", c$
PRINT "El caracter introducido es ";
COLOR 10

IF c$ = "1" XOR c$ = "2" XOR c$ = "3" XOR c$ = "4" XOR c$ = "5" THEN
   PRINT "número"
ELSE
   IF c$ = "+" XOR c$ = "-" XOR c$ = "*" XOR c$ = "/" XOR c$ = "\" XOR c$ = "^" THEN
      PRINT "operador  matemático"
   ELSE
      IF c$ = "a" XOR c$ = "b" XOR c$ = "c" XOR c$ = "d" XOR c$ = "e" THEN
         PRINT "letra minúscula"
      ELSE
         IF c$ = "A" XOR c$ = "B" XOR c$ = "C" XOR c$ = "D" XOR c$ = "E" THEN
            PRINT "letra mayúscula"
         ELSE
            PRINT "otro caracter diferente de letra (a – e), número (1 – 5) u operador matemático"
         END IF
      END IF
   END IF
‘ aquí falta un END IF
```

```
END
```

QuickBASIC nos marcará el error "*bloque IF sin END IF*"



Una cosa que puedes hacer para evitar este error es escribir el bloque completo y dentro de este ir escribiendo las instrucciones que se vayan necesitando.

SELECCIÓN MÚLTIPLE: SELECT CASE

Aquí utilizaremos una sentencia nueva: el bloque **SELECT CASE**. El bloque SELECT CASE nos sirve para seleccionar de entre múltiples valores que pudiera tener una expresión.
Veamos el siguiente ejemplo:

```
CLS
PRINT "1) Opción 1"
PRINT "2) Opción 2"
PRINT "3) Opción 3"
PRINT
INPUT "Escoge una opción -> ", op
SELECT CASE op                          ‘ Seleccionamos valor de op:
  CASE IS = 1                           ‘ Si la opción introducida es la 1...
    PRINT "Escogiste la opción 1"
  CASE IS = 2                           ‘ Si la opción introducida es la 2...
    PRINT "Escogiste la opción 2"
  CASE IS = 3                           ‘ Si la opción introducida es la 3...
    PRINT "Escogiste la opción 3"
  CASE ELSE                             ‘ Si la opción no es ninguna de las anteriores...
    PRINT "Escogiste una opción no válida"
END SELECT                              ‘ Fin de seleccionar valor
END
```

El ejemplo anterior nos presenta un pequeño menú con 3 opciones a escoger, luego selecciona los posibles valores que pudiera tener la variable ‘op’ (SELECT CASE op); si escogemos la opción 1, y por tanto le damos a la variable ‘op’ el valor de uno, entonces se ejecutaron las instrucciones que corresponden al **caso** de que la variable sea igual a 1 (CASE IS = 1), si escogemos la opción 2, entonces se ejecutarán las instrucciones que corresponden al caso de que la variable sea igual a 2 (CASE IS = 2), etc.

Dentro de cada caso que tenga la expresión podemos ejecutar una o más instrucciones, incluyendo bloques IF...THEN...ELSE y anidar bloques SELECT CASE si nos fuera necesario.

```
REM Los comentarios de las edades son a mi particular parecer y si se asemejan a la realidad
REM son mera coincidencia :-)
CLS
DIM edad AS INTEGER
PRINT
```

```
INPUT "Introduce tu edad -> ", edad

SELECT CASE edad
  CASE IS <= 9
    PRINT "¡Eres un niño!"
  CASE IS <= 12
    PRINT "¡Eres un puberto!"
  CASE IS <= 18
    PRINT "Eres un adolescente"
  CASE IS <= 30
    PRINT "Eres un joven"
  CASE IS <= 45
    PRINT "Ya eres un adulto”
  CASE IS <= 60
    PRINT "Ya has vivido muuucho"
  CASE IS <= 80
    PRINT "Viejos los cerros..."
  CASE IS < 100
    PRINT "Sin comentarios..."
  CASE IS >= 100
    PRINT "Tal vez…"
END SELECT
```

El caso ELSE (CASE ELSE) es opcional.

Capítulo 3: Sentencias de control repetitivas

EL BLOQUE WHILE... WEND

Ahora veamos como podemos **repetir** partes de un programa **mientras** que cierta condición se cumpla o sea **verdadera**...

```
' Ejemplo de WHILE... WEND
' Calcula áreas de 5 círculos

CONST pi = 3.1416

CLS
PRINT TAB(30); "AREA DE 5 CIRCULOS"
PRINT
contador = 0   ' Inicializamos el contador

WHILE contador < 5                'Inicio del ciclo. Contador < 5 porque empezamos en cero
   INPUT "Radio -> ", radio
   PRINT "El área es"; pi * radio * radio
   PRINT
   contador = contador + 1        ' El contador se incrementa en 1
WEND
END
```

El bloque **WHILE** ("mientras")... **WEND** nos sirve para que mientras una condición se cumpla (en este caso, mientras *contador < 5*) repetir una serie de instrucciones, desde donde empieza el WHILE hasta que se encuentre el WEND. En este tipo de ciclos o bucles generalmente **el valor de la condición ya ha sido inicializado** desde antes de entrar al ciclo (*contador = 0*) y **se incrementa** dentro del ciclo (*contador = contador + 1*). El ciclo se ejecutará mientras la condición sea verdadera, por lo que si desde antes de entrar al bucle la condición ya es falsa, el programa no ejecutará el ciclo. Veamos...

```
' Ejemplo de un bucle WHILE...WEND que no se ejecutará

i = 0
CLS
WHILE i = 1   ' Mientras i = 1...pero como i = 0 y no igual a 1...
   PRINT "HOLA"
   i = i + 1
WEND
PRINT "Fin del programa"
END
```

Nota: Si lo ejecutas paso a paso <F8> en QuickBasic verás mejor la secuencia.

También podemos caer en **ciclos infinitos**; esto es, ciclos que no tengan fin.

```
' Ejemplo de un bucle WHILE...WEND infinito
i = 1                             ' Inicializamos la variable
CLS
```

```
WHILE i = 1                          ' Mientras i = 1...
   PRINT "HOLA, MUNDO..."
WEND                                  '...
PRINT "Fin del programa"
END
```

Nota: Para poder salir de un bucle infinito debemos utilizar la combinación de teclas CTRL + Pause o CTRL + Scroll. En Turbo Basic asegúrate de la opción *Keyboard Break* del menú *Options* este en ON.

¿Qué podemos observar en el programa anterior? En efecto el programa entrará en el ciclo ya que la condición es verdadera (*i = 1*), pero *i* siempre será igual a 1 ya que no hay nada que la modifique dentro del bucle y así poder dar una salida. Este tipo de errores suele ocurrir si por ejemplo...

```
' Ejemplo de un bucle WHILE...WEND infinito por error interno
cont = 1                     ' Inicializamos la variable

CLS
WHILE cont = 1                       ' Mientras cont = 1...
   PRINT "HOLA, MUNDO..."
   cont = con + 1                    ' ¿¿ con ??. Si con = 0, entonces cont = 0 + 1 = 1 (:-O)
WEND                                 ' ...
PRINT "Fin del programa"
END
```

También no necesariamente tiene que ser un incremento el que nos dé la salida...

```
' Ejemplo de un bucle WHILE...WEND utilizando un método de salida diferente
salir = 0                            ' Inicializamos la variable

CLS
WHILE salir = 0                      ' Mientras salir = 0...
   PRINT "HOLA, MUNDO..."
   INPUT "¿Quieres salir (si = 1, no = 0)? ", salir
   PRINT
WEND                                 ' seguir.
PRINT "Fin del programa"
END
```

Si usamos un contador, también podemos incrementarlo como queramos...

```
' Sacando raíz cuadrada aproximada usando WHILE...WEND.

CLS
PRINT TAB(26); "CALCULO DE LA RAIZ CUADRADA APROXIMADA"
PRINT
INPUT "Introduce un número positivo -> ", n
IF n < 0 THEN END
WHILE (cont * cont) < n
   cont = cont + .0001
```

```
WEND

PRINT "La raíz cuadrada aproximada de"; n; "es"; cont
END
```

- No necesitamos declarar una variable *cont = 0* ya que al crearla automáticamente asume este valor.
- Si *n* llegara a ser negativo, el programa se acaba.

EL BLOQUE DO...LOOP

El bloque **DO** ("hacer")...**LOOP** ("bucle") tiene 4 variantes. Veamos la primera:

DO WHILE...LOOP

Este tipo de bloque es muy parecido a WHILE...WEND, ya que la condición regularmente se conoce por adelantado, y existe la posibilidad de que nunca se ejecute el ciclo. **Mientras** la condición sea **verdadera**, el ciclo se ejecutará.

Sintaxis:

```
DO WHILE <condición>
  instrucción1
  instrucción2
  instrucción3
  [EXIT DO]
  ....
  instrucciónn
LOOP
```

Ejemplo:

```
' Ejemplo de DO WHILE...LOOP

i = 0            ‘ Inicializamos la variable
PRINT "El valor de i al empezar el ciclo es”; i
DO WHILE I < 10
  I = I + 1
  ? “Luego I vale”; I ;”...”
LOOP
END
```

Cuando LOOP encuentra a la condición falsa se termina el ciclo. Si no quisiéramos que se ejecutara el ciclo...

```
‘ Ejemplo de DO WHILE...LOOP que nunca se ejecutará
CLS
i = 1                                ‘ Inicializamos la variable
PRINT “Iniciamos el programa...”
DO WHILE i < 1                       ‘¿Es i < 1?
  PRINT “Estamos dentro del ciclo”
```

```
  i = i + 1
LOOP
PRINT “fin del programa”
END
```

DO...LOOP WHILE

En este tipo de ciclos las instrucciones se ejecutarán **por lo menos 1 vez**, ya que la condición se evalúa cuando se termina de ejecutar el bloque de instrucciones. Se ejecutara el ciclo **mientras** la condición sea **verdadera**.

Sintaxis:

```
DO
  instrucción1
  instrucción2
  instrucción3
  [EXIT DO]
 ....
  instrucciónn
LOOP WHILE <condición>
```

Veamos el siguiente ejemplo:

```
' Ejemplo de DO...LOOP WHILE. Calculo de la velocidad de un vehículo con MRU

CLS
COLOR 10        ‘ Verde brillante
PRINT TAB(20); "CALCULO DE LA VELOCIDAD DE UN VEHICULO"
COLOR 7: PRINT          ‘ Color gris normal
DO          ‘ Hacer...
  INPUT "Distancia recorrida por el vehículo (m) -> ", d
  INPUT "Tiempo en recorrerla (s) -> ", t
  v = d / t
  PRINT "La velocidad con que viajó el vehículo es"; v; "m / s"
  PRINT
  INPUT "¿Desea calcular otra vez (si = 1, no = 0)"; desea
LOOP WHILE desea = 1    ‘Mientras desea = 1
END
```

Nota: La sentencia EXIT DO nos permite salir de un bloque DO en cualquier momento.

DO UNTIL...LOOP:

La palabra UNTIL (“hasta que”) nos dice que mientras la condición **NO** sea verdadera (a diferencia de WHILE), el bucle se ejecutará.

Sintaxis:

```
DO UNTIL <condición>            ' “Hacer hasta que” condición
  instrucción1
  instrucción2
```

```
   instrucción3
   [EXIT DO]
   ....
   instrucciónn
LOOP                                    ' "Bucle"
```

Veamos el siguiente ejemplo:

```
' Ejemplo de DO UNTIL...LOOP

CLS
COLOR 10
PRINT TAB(28); "AREA DE TRIANGULOS"
COLOR 7: PRINT : PRINT
INPUT "Cuántas areas va a calcular -> ", n
PRINT
DO UNTIL c = n
   INPUT "Altura -> ", b
   INPUT "Base-> ", h
   a = (b * h) / 2
   PRINT "El area es"; a
   PRINT
   c = c + 1
LOOP
END
```

Al crearse la variable *c* (contador) esta asume un valor de 0 por lo que:

- Si el número de areas a calcular (*n*) es 0, entonces la condición se hace verdadera antes de entrar al ciclo (en efecto 0 = 0) y por tanto el ciclo no se ejecutará.
- Si *n* es positiva el ciclo que se ejecutará n veces hasta que la condición sea verdadera.
- Si *n* es negativa el bucle se hace infinito, ya que cuando incrementamos el contador alejamos más a c de ser un numero negativo igual a n.

Veamos un ejemplo más...

```
' Ejemplo de DO UNTIL..LOOP. Una manera de hacer cuentas.
salir = 0

CLS
COLOR 10
PRINT TAB(30); "ABARROTES LA CHONA"
PRINT TAB(20); "C. NOGALES #2500-B, COL. MIRADOR"
COLOR 7: PRINT : PRINT
PRINT "INTRODUZCA 0 PARA TERMINAR"
PRINT

DO UNTIL salir = 1              ' Hacer hasta que salir = 1
   COLOR 7
   PRINT "Precio del producto $ "; : COLOR 15: INPUT "", prod
   IF prod = 0 THEN salir = 1
   suma = suma + prod
```

```
LOOP                        ' Bucle
PRINT : COLOR 7
PRINT "Total  $ "; : COLOR 15: PRINT suma
END
```

DO... LOOP UNTIL

En este tipo de bucles, las sentencias se ejecutarán **al menos una vez** y **hasta** que la condición sea **verdadera**.

Sintaxis:

```
DO                                  'Hacer...
   instrucción 1
   instrucción2
   instrucción3
   ....
   [EXIT DO]
   instrucciónn
LOOP UNTIL <condición>        ' Hasta que condición sea verdadera.
```

Ejemplo...

```
' Ejemplo de DO...LOOP UNTIL. Calcula la energía cinética de un cuerpo en movimiento.

DO
   ' Cabeceras
   CLS
   COLOR 10
   PRINT TAB(25); "CALCULO DE LA ENERGÍA CINÉTICA"
   PRINT : PRINT : COLOR 7

   ' Entrada de datos
   INPUT "Introduzca la masa del cuerpo (kg) -> ", m
   INPUT "Velocidad con que viaja (m / s) -> ", v

   ' Cálculo del resultado
   ec = (m * (v ^ 2)) / 2  ' Fórmula de la energía cinética

   ' Salida del resultado
   PRINT "La energía cinética del cuerpo es";
   COLOR 15: PRINT ec;           ' Color blanco brillante
   COLOR 7: PRINT "Newtons-metros (joules)"

   ' Ver si otro cálculo
   PRINT
   INPUT "Otro cálculo (si = 1, no = 0) -> ", otro
LOOP UNTIL otro = 0
END
```

DO y LOOP pueden o no contener un WHILE o un UNTIL. Si consultas la ayuda de QuickBASIC a cerca de la orden DO obtendrás la siguiente sintaxis:

Sintaxis 1:

```
DO [{WHILE | UNTIL} condición]
  [instrucción]
  [EXIT DO]
LOOP
```

Sintaxis 2:

```
DO
  [instrucción]
  [EXIT DO]
LOOP [{WHILE | UNTIL} condición]
```

Veamos el método que se utiliza en la ayuda que viene con QB para describir la sintaxis de una sentencia:

- Las palabras en mayúsculas son palabras reservadas del lenguaje QB.
- Los corchetes (**[ ]**) indican que lo que hay dentro de estos es opcional.
- La barrita (**|**) y las llaves (**{ }**) significan que podemos utilizar –según nuestro uso– solo una de las ordenes que van entre las llaves y que van separadas por la barrita; no pueden ir dos de estas sentencias juntas, por ejemplo: DO WHILE UNTIL <condición>.
- Por ***sentencia*** o ***instrucción*** entenderemos de aquí en delante: una sola sentencia o instrucción, o también una serie de sentencias o instrucciones.

Siguiendo con lo de la sentencia DO, esta puede ir sola junto con LOOP. Lo que hace esta sentencia es ejecutar un ciclo infinito ya que aparentemente no hay nada que lo detenga al no tener una condición que proporcione la salida. Se puede detener el ciclo al ejecutar el programa con la combinación de teclas CTRL + PAUSE o CRTL + SCROLL. Pero...¿cómo hacer para que se detenga sin usar este método externo?. Para salir de cualquier bloque DO utilizamos la sentencia **EXIT DO**. Veamos el siguiente ejemplo...

```
' Ejemplo de DO...LOOP sin salida
' 25/09/2000

CLS
DO
  PRINT "¡QuickBASIC es fácil!"
LOOP
END
```

Este tipo de bloque como podemos ver es infinito y solo con CTRL + SCROOL podemos salir. Para salir de un bloque DO podemos usar EXIT DO:

```
' Ejemplo de DO..LOOP con salida

DO
  CLS
  COLOR 10
```

```
PRINT TAB(29); "MENÚ DE OPERACIONES"
PRINT TAB(27); "-----------------------"
COLOR 7
PRINT TAB(32); "1) SUMAR"
PRINT TAB(32); "2) RESTAR"
PRINT TAB(32); "3) MULTIPLICAR"
PRINT TAB(32); "4) DIVIDIR"
PRINT TAB(32); "5) EXPONENCIAR"
PRINT TAB(32); "6) RAIZ CúBICA APROXIMADA"
PRINT TAB(32); "7) SALIR"

PRINT : PRINT
PRINT TAB(30); : INPUT "ESCOGA SU OPCIÓN -> ", opcion
CLS

SELECT CASE opcion
  CASE IS = 1  ' Sumar
    COLOR 10
    PRINT TAB(38); "SUMAR"
    PRINT TAB(36); "---------"
    PRINT : COLOR 7
    INPUT "Primer número -> ", n1
    INPUT "Segundo número -> ", n2
    PRINT "La suma de"; n1; "y"; n2; "es"; n1 + n2
  CASE IS = 2  ' Restar
    COLOR 10
    PRINT TAB(37); "RESTAR"
    PRINT TAB(36); "--------"
    PRINT : COLOR 7
    INPUT "Primer número -> ", n1
    INPUT "Segundo número -> ", n2
    PRINT "La resta de"; n1; "menos"; n2; "es"; n1 - n2
  CASE IS = 3   ' Multiplicar
    COLOR 10
    PRINT TAB(34); "MULTIPLICAR"
    PRINT TAB(32); "---------------"
    PRINT : COLOR 7
    INPUT "Primer número -> ", n1
    INPUT "Segundo número -> ", n2
    PRINT "El producto de"; n1; "por"; n2; "es"; n1 * n2
  CASE IS = 4  ' Dividir
    COLOR 10
    PRINT TAB(36); "DIVIDIR"
    PRINT TAB(35); "---------"
    PRINT : COLOR 7
    INPUT "Dividendo -> ", n1
    INPUT "Divisor -> ", n2
    PRINT "El cociente de"; n1; "entre"; n2; "es"; n1 / n2
  CASE IS = 5  ' Exponenciar
    COLOR 10
    PRINT TAB(34); "EXPONENCIAR"
    PRINT TAB(33); "-------------"
    PRINT : COLOR 7
    INPUT "Base -> ", n1
    INPUT "Potencia -> ", n2
    PRINT "El resultado de elevar"; n1; "a la"; n2; "es"; n1 ^ n2
```

```
    CASE IS = 6  ' Radicar
      COLOR 10
      PRINT TAB(34); "RAIZ CÚBICA APROXIMADA"
      PRINT TAB(33); "-------------"
      PRINT : COLOR 7
      INPUT "Número para sacarle raíz cúbica aproximada-> ", n1
      DO WHILE (cont * cont * cont) < n1
        cont = cont + .001
      LOOP
      PRINT "La raíz cúbica aproximada de"; n1; "es"; cont
    CASE IS = 7
      EXIT DO  'Sale del bloque DO
    CASE ELSE
      COLOR 18   'Verde azulado brillante intermitente
      PRINT "--OPERACIÓN NO VALIDA--"
  END SELECT
  PRINT: COLOR 7
  INPUT "PRESIONE <ENTER> PARA CONTINUAR...", enter$
LOOP
END
```

READ-DATA

Esta estructura nos permite cargar una o mas variables, con valores previamente almacenados con la sentencia DATA. Veamos el siguiente ejemplo:

```
CLS
'Lee la primera DATA
READ nombre$, sem$, edad$
PRINT "Alumno: "; nombre$
PRINT "Semestre: "; sem$
PRINT "Edad: "; edad$
PRINT
'Lee la segunda DATA
READ nombre$, sem$, edad$
PRINT "Alumno: "; nombre$
PRINT "Semestre: "; sem$
PRINT "Edad: "; edad$

DATA "Pablo Jorge Ruiz Alarcón", "III", 22
DATA "Luis José Perez Hurtado", "IV", 19
```

La sentencia **RESTORE** hace que se vuelven a leer los datos desde la primera DATA

```
CLS
'Lee la primera DATA
READ nombre$, sem$, edad$
PRINT "Alumno: "; nombre$
PRINT "Semestre: "; sem$
PRINT "Edad: "; edad$

PRINT
```

```
RESTORE
'Vuelve a leer la primera DATA
READ nombre$, sem$, edad$
PRINT "Alumno: "; nombre$
PRINT "Semestre: "; sem$
PRINT "Edad: "; edad$

DATA "Pablo Jorge Ruiz Alarcón", "III", 22
DATA "Luis José Perez Hurtado", "IV", 19
```

Si se intentan almacenar valores de cadena en variables numéricas se produce un error. Igualmente si se intenta leer después de que se han pasado todas las DATA y no haya un RESTORE para reiniciar desde el principio.

EL CICLO FOR..NEXT

Otra estructura muy útil cuando trabajamos con ciclos es el **FOR...NEXT**. Su sintaxis es la siguiente:

Sintaxis:

```
FOR contador = valor inicial TO valor final [STEP incremento]
  <sentencia>
  [EXIT FOR ]
NEXT [contador]
```

La sentencias se ejecutan hasta que el contador llegue a su valor final; si omitimos la palabra **STEP** el contador se incrementa por default en 1. EXIT FOR sirve para salir del bloque FOR en cualquier momento. Veamos el siguiente ejemplo:

```
' Ejemplo de FOR...NEXT

CLS
COLOR 10
PRINT TAB(30); "ÁREA DE TRÍANGULOS"
PRINT : COLOR 7
INPUT "CUÁNTAS ÁREAS DESEA CALCULAR -> ", n
PRINT
FOR contador = 1 TO n
  COLOR 7
  PRINT "TRIÁNGULO #"; : COLOR 15: PRINT contador
  INPUT "Base -> ", b
  INPUT "Altura -> ", h
  a = (b * h) / 2
  PRINT "El área es"; : COLOR 15: PRINT a
  PRINT
NEXT contador
END
```

Este tipo de bucle se ejecuta **contador final – contador inicial + 1** veces. Viendo el ejemplo anterior, podemos comprobar que el bucle se ejecutará 3 – 1 + 1 = 3 veces. Veamos este otro programa:

```
' Imprime las tablas de multiplicar. Usa FOR
' 25/09/2000

CLS
INPUT "¿Cuál tabla de multiplicar desea ver"; tabla
PRINT
FOR c = 0 TO 10
   PRINT tabla; " X "; c; " = "; c * tabla
NEXT
END
```

Tambien podemos tener ciclos FOR anidados, en donde cada FOR se termina con el NEXT que le queda mas cerca…

```
' Imprime los colores que usamos en QuickBASIC. Usa FOR

CLS
FOR texto = 0 TO 31        ' Los colores de texto van del 0 al 31
   FOR fondo = 0 TO 7      ' Los colores de fondo van del 0 al 7
      COLOR texto, fondo
      PRINT "Color"; texto; fondo
   NEXT  fondo
NEXT texto
END
```

En el ejemplo anterior podemos notar que:

- Para cuando el primer FOR (texto) dé una vuelta, el FOR siguiente (fondo) ya terminó su ciclo.
- Podemos usar variables para establecer los valores de los colores (sentencia COLOR) así como de cualquier otra sentencia que utilice valores. Por ejemplo TAB(n), donde n es un variable que obviamente contiene un numero que se utilizará para indicarle n espacios a TAB.

No solo podemos utilizar FOR para incrementar una variable, también podemos **decrementarla** utilizando **STEP** con la cantidad en que queremos disminuirla:

```
' imprime los números del 100 al 0, vía FOR
' 26/09/2000
inicio = 100
fin = 0
decremento = -1

CLS
PRINT "Imprime los números del 100 al 0, vía FOR"
PRINT
INPUT "PRESIONE <ENTER> PARA COMENZAR...", enter$  ‘ Para que el programa se detenga
CLS
```

```
FOR c = inicio TO fin STEP decremento
   PRINT c
NEXT c
PRINT "Fin del programa."
END
```

El ciclo de tipo FOR...NEXT es muy útil para llenar **arreglos**, los cuales son estructuras de datos que veremos en el siguiente capitulo...

Capítulo 4: Funciones matemáticas

CONCEPTO MATEMÁTICO DE FUNCIÓN

Primero que nada, es necesario tener una noción de lo que es en sí una función. Una función puede ser definida como una **correspondencia** entre dos variables, una llamada variable dependiente y otra llamada variable dependiente. Veamos la siguiente notación:

$$y = f(x)$$

Donde:
y : Es la variable dependiente de la función. Se le llama dependiente porque para que pueda tomar un valor, **depende** de los valores que pueda tomar la variable x. También podemos decir que "y esta en función de x".
x: Es la variable independiente de la función. Se dice que es independiente ya que puede tomar los valores que quiera.
f: Se puede decir que es el procedimiento o ecuación que tomará a x para devolverle un valor a y.

Veamos una función muy común:

$$y = x^2 + x$$

La función anterior tiene a ***y*** como variable dependiente y a ***x*** como variable independiente, y el procedimiento es elevar a ***x*** (variable independiente) al cuadrado y sumarle ***x***. Por cada valor que tome ***x***, a ***y*** le corresponderá otro.

FUNCIONES MATEMÁTICAS Y ARITMETICAS

Ahora veamos algunas de las más comunes funciones matemáticas que son parte del lenguaje BASIC (funciones predefinidas):

ABS(x)
Devuelve el valor sin signo de x (valor absoluto). Ejemplo:

```
CLS
INPUT "Introduce dos números (a, b)-> ", a, b
PRINT "El valor absoluto de "; a; "es"; ABS(a); "y el de "; b; "es"; ABS(b) : END
```

SIN (x)
Devuelve el seno de x, siendo x el valor de un ángulo dado en **RADIANES**. Ejemplo:

```
' Calcula la componente vertical de la velocidad de un misil balístico
CONST pi = 3.1416

CLS
INPUT "Introduzca la velocidad (m / s) del misil -> ", v
INPUT "Introduzca el ángulo (en grados) del lanzamiento -> ", grados
radianes = grados * pi / 180
y = v * SIN(radianes)
PRINT "La velocidad vertical del misil es de"; vy; "m / s"
```

```
END
```

Nota: Como son muy comunes las conversiones de grados a radianes, para hacer esto se utiliza la fórmula:

angulo_en_radianes = (angulo_en_grados x 3.1416) / 180

COS (x)
Devuelve el coseno de x, donde x es el valor de un ángulo en radianes. Ejemplo:

```
' Calcula la componente en el eje x de la velocidad del mismo misil
CONST pi = 3.1416

CLS
INPUT "Introduzca la velocidad (m / s) del misil -> ", v
INPUT "Introduzca el ángulo (en grados) del lanzamiento -> ", grados
vx = v * COS(grados * pi / 180)
PRINT "La velocidad horizontal del misil es de"; vx; "m / s"
END
```

TAN (x)
Devuelva la tangente de x, siendo x el valor de un ángulo en radianes. Ejemplo:

```
CONST PI = 3.1416

CLS
INPUT "Introduce el ángulo de una recta -> ", grados
PRINT "Tangente de la recta: "; : COLOR 15: PRINT TAN(grados * PI / 180)
PRINT : COLOR 7
INPUT "Presione <ENTER> para continuar...", enter$
END
```

ATN(x)
Devuelve el ángulo equivalente a la tangente x.

```
' Calcula el ángulo del misil a partir de los componentes de la velocidad
CONST PI = 3.1416

CLS
INPUT "Introduzca la velocidad horizontal -> ", vx
INPUT "Introduzca la velocidad vertical -> ", vy

' Si la componente en x de la velocidad es 0 se producirá un error ya
' que estaremos dividiendo entre 0. Esto lo puedes arreglar con un IF
v = vy / vx
radianes = ATN(v) ' ATN devolverá un resultado en RADIANES
grados = (radianes * 180) / PI

PRINT "El ángulo (en grados) del lanzamiento es: "; : COLOR 15: PRINT grados
PRINT : COLOR 7
```

```
INPUT "Presione <ENTER> para continuar...", enter$
END
```

LOG(x)
Devuelve el logaritmo base e (logaritmo natural) de x, y donde la constante e = 2. 7182 y x es un valor MAYOR que cero.

```
CLS
PRINT "El logaritmo natural de 9 es "; LOG(9)
END
```

El logaritmo base e se escribe como ***log*** $_e$ o como ***ln***.

EXP(x)
Devuelve e a la x, donde x es un valor menor o igual a 88.02969. A esta función se le llama antilogaritmo en base e.

```
CLS
n = 9
PRINT "e elevado a la potencia "; n ;" da "; EXP(n)
END
```

Comprendiendo lo hecho: En matemáticas, un **logaritmo** se define como la potencia o exponente a la que se ha de elevar un numero fijo (llamado base) para dar como resultado un numero dado:

$$\ln 9 \approx (2.7182)^{2.197225}$$

➔ Podemos decir que si nos dan 9 para sacar su logaritmo natural (***e*** es la **base**), entonces el resultado será 2.197225.

Por otro lado, un **antilogaritmo** se define como el resultado de elevar una base a un número dado.

➔ Tomando el ejemplo anterior podemos decir que si nos dan 2.197225 para sacar su antilogaritmo natural, entonces el resultado será 9.

Las siguientes funciones marcadas con asterisco son exclusivas de Turbo Basic (de Borland).

*** LOG2(x)**
Da como resultado el logaritmo base 2.

```
cls
n = 32
? "El logaritmo base 2 de "; n ; "es"; log2(n)
end
```

* **LOG10(x)**
Da como resultado el logaritmo base 10.

```
cls
n = 1000
? “El logaritmo base 10 de “; n ; ”es”; log10(n)
end
```

* **EXP2(x)**
Devuelve 2 a la x.

```
cls
n = 2
? “El resultado de elevar 2 a la ”; n ;”es “; exp2(n)
end
```

* **EXP10(x)**
Devuelve 10 a la x.

```
cls
n = 3
? “El resultado de elevar 10 a la “; n ;” es “; exp10(n)
end
```

SWAP x, y
Intercambia los valores de las variables.

```
x = 5 : y = -3
CLS
COLOR 15 : PRINT "Primero:"
COLOR 7: PRINT "x = "; x ;"  y = ";y
COLOR 15 : PRINT "Luego:"
COLOR 7: PRINT "x = "; x ;"  y = ";y
END
```

FIX(x)
Le quita los decimales a x, siendo x un numero real.

```
CLS
n = 3.1416
m = 2.90
PRINT “Si le quitamos los decimales a “; n ;” el resultado es: ”; FIX(n)
PRINT “Si le quitamos los decimales a “; m ;” el resultado es: ”; FIX(m)
END
```

Nota: La función FIX trunca un entero, NO lo redondea.

INT(x)
Redondea *x* al número menor inmediato que sea menor o igual a x.

```
n = 3.65 : m = 2.3
o = -3.2 : p = 1
CLS
PRINT "Si redondeamos con INT a "; n; "el resultado es "; INT(n)
PRINT "Si redondeamos con INT a "; m; "el resultado es "; INT(m)
PRINT "Si redondeamos con INT a "; o; "el resultado es "; INT(o)
PRINT "Si redondeamos a con INT "; p; "el resultado es "; INT(p)
END
```

SQR(x)
Calcula la raíz cuadrada de un número o expresión mayor o igual a cero. El resultado se da en doble precisión.

```
' Obtiene la hipotenusa de un triángulo rectángulo a partir de los catetos
CLS : COLOR 15
PRINT TAB(15); " CÁLCULO DE LA HIPOTENUSA A PARTIR DE LOS CATETOS"
PRINT : PRINT : COLOR 7
INPUT "Cateto opuesto -> ", cop
INPUT "Cateto adyacente -> ", cad

hip = SQR((cop * cop) + (cad * cad))

PRINT
PRINT "La hipotenusa vale "; : COLOR 15: PRINT hip
PRINT : COLOR 7
INPUT "PRESIONE <ENTER> PARA CONTINUAR", enter$
END
```

SGN(x)
La función SGN trabaja de la siguiente forma:

- Si x es **negativo**, SGN devuelve **–1**.
- Si x es **cero**, SGN devuelve **0**.
- Si x es **positivo**, SGN devuelve **1**.

El siguiente ejemplo resuelve ecuaciones cuadráticas del tipo $ax^2 + bx + c = 0$ utilizando la fórmula

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

'Resuelve ecuaciones cuadráticas del tipo $ax^2 + bx + c = 0$ por la fórmula gral.

```
DO
  CLS : COLOR 15
  PRINT TAB(15); " RESUELVE ECUACIONES CUADRATICAS POR LA FORMULA GENERAL"
  PRINT TAB(35); "-b *
  PRINT : PRINT : COLOR 7
  INPUT "Introduce el valor de a -> ", a
  INPUT "Introduce el valor de b -> ", b
```

```
  INPUT "Introduce el valor de c -> ", c
  PRINT : COLOR 7

  discriminante = b * b - 4 * a * c

  SELECT CASE SGN(discriminante)
    CASE IS = -1
      COLOR 16
      PRINT "La ecuación no tiene una solución en los reales."
      'No se le puede sacar raíz cuadrada a un número negativo
   CASE IS = 0
      PRINT "La ecuación tiene una solución:"
      ' Cero da lo mismo sumado que restado.
      x = -b / (2 * a)
      PRINT "x1= "; x; "y x2= "; x
    CASE IS = 1
      PRINT "La ecuación tiene dos soluciones:"
      x1 = (-b + SQR(discriminante)) / (2 * a)
      x2 = (-b - SQR(discriminante)) / (2 * a)
      PRINT "x1 = "; x1
      PRINT "x2 = "; x2
  END SELECT
  PRINT
  INPUT "Quieres realizar otro cálculo (S\N)"; otro$
LOOP UNTIL UCASE$(otro$) = "N"     ' UCASE$(c$) devuelve a c$ en mayúsculas
END
```

NÚMEROS ALEATORIOS

RND

La función RND devuelve un número (tipo Single) al azar entre el rango de 0 y 1. Cada vez que se corra el programa, RND generará los mismos números "aleatorios", por lo que se requiere de un generador para que cada vez que se corra el programa los números sean diferentes.

```
' Imprime números al azar en el rango de 0 a 32

n = 32

CLS
FOR i = 1 TO 10
  PRINT INT(RND * (n + 1))
NEXT i
END
```

TIMER

Esta función retorna los segundos transcurridos desde la media noche de la maquina. TIMER devuelve un valor real en el intervalo abierto entre 0 y 86400.

```
CLS
PRINT "Han trascurrido"; TIMER; "segundos desde la media noche de la maquina"
```

Sentencia RANDOMIZE [n]
Inicializa el generador de números aleatorios a partir de una semilla (n). Si no se da una semilla, QuickBASIC detendrá el programa y le pedirá una al usuario; por otro lado, si esta semilla permanece constante durante el programa, los números generados serán los mismos cada vez que se corra el programa. Esta sentencia se usa para generar números aleatorios para la función RND.

Para evitar que se "atasquen" los números aleatorios, es muy común el uso de la función TIMER para darle semillas diferentes a RANDOMIZE. Pero antes veamos el siguiente ejemplo:

```
CLS
RANDOMIZE
PRINT "Un número al azar -->> "; RND
END
```

Como no utilizamos ninguna semilla para RANDOMIZE, QB preguntará por un número para iniciar el generador de números aleatorios. Veamos este otro:

```
CLS
RANDOMIZE 4 : PRINT "Un número al azar -->> "; RND
END
```

Como la semilla es siempre la misma en todo el programa, cada vez que este se corra, el número "al azar" será siempre el mismo. Por lo tanto si queremos números que no se repitan debemos usar una semilla que cambie cada vez que se inicie el programa:

```
CLS
RANDOMIZE TIMER
PRINT "Un número al azar -->> "; RND
END
```

FUNCIONES DE CONVERSIÓN NUMÉRICA

CINT(x)
Redondea x al entero más cercano, siendo x un número con decimales que está en el rango de los enteros cortos (-32768 a 32767). Si x esta fuera del rango de los enteros cortos, entonces se produce un mensaje de "overflow" (desbordamiento).

```
n = 3.65
m = 2.3
o = -3.2
p = 1

CLS
PRINT "Si redondeamos a "; n; "el resultado es "; CINT(n)
PRINT "Si redondeamos a "; m; "el resultado es "; CINT(m)
PRINT "Si redondeamos a "; o; "el resultado es "; CINT(o)
```

```
PRINT "Si redondeamos a "; p; "el resultado es "; CINT(p)
END
```

CLNG(x)

Redondea x al entero más cercano, siendo x un numero con decimales que está en el rango de los enteros largos (-2,147,483,648 a 2,147,483,647). Si x esta arriba del rango de los enteros largos, entonces se produce un mensaje de "overflow" (desbordamiento).

```
a=345767.45
b=122767.55
PRINT a, CLNG(A)
PRINT b, CLNG(B)
END
```

CSNG(x)

Da una **precisión simple** a x, siendo x de tipo Double, pero estando en el rango de los SINGLE (1.5 E-45 a 3.4 E38).

```
a# = 5.6634537105#
b# = 6.4598755434#

CLS
COLOR 15
PRINT "Doble precisión", "Simple precisión"
COLOR 7
PRINT a#, CSNG(a#)
PRINT b#, CSNG(b#)
END
```

CDBL(x)

Da una **precisión doble** a x, teniendo x una precisión simple y estando x dentro del rango de los Double (5.0e-324 a 1.7e308).

```
a = 5.6634 : b = 6.4598

CLS
COLOR 15
PRINT "Simple precisión", "Doble precisión"
COLOR 7
PRINT a; : PRINT TAB(29); CDBL(a)
PRINT b; : PRINT TAB(29); CDBL(b)
END
```

INTRODUCCION A LOS ARREGLOS

ARREGLOS DE UNA DIMENSION

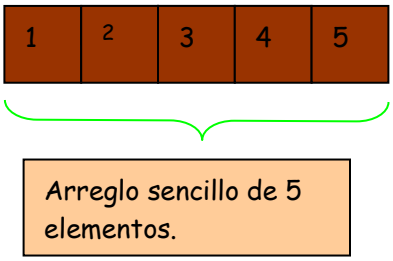
Hasta el momento hemos visto las estructuras de datos más sencillas que son las variables, pero ahora veremos otras estructuras de datos que se llaman **arreglos**.

Para ver más o menos para que se utilizan estas estructuras, pensemos en lo siguiente: Debemos introducir los nombres de 5 personas y guardarlos para luego volverlos a imprimir en el mismo orden en que entraron, ¿cómo se haría esto?. La primera forma que se nos podría ocurrir (ya que no sabemos nada sobre arreglos, en teoria) es usar 5 variables de tipo String, guardar los nombres en ellas y luego volverlos a imprimir, algo así como...

```
CLS
PRINT : COLOR 15
PRINT "CAPTURA DE PERSONAS:"
COLOR 7
INPUT "Persona 1 ->"; persona1$
INPUT "Persona 2 ->"; persona2$
INPUT "Persona 3 ->"; persona3$
INPUT "Persona 4 ->"; persona4$
INPUT "Persona 5 ->"; persona5$

PRINT : COLOR 15
PRINT "SALIDA DE PERSONAS:"
COLOR 7
PRINT "Persona 1 ->"; persona1$
PRINT "Persona 2 ->"; persona2$
PRINT "Persona 3 ->"; persona3$
PRINT "Persona 4 ->"; persona4$
PRINT "Persona 5 ->"; persona5$
END
```

Esta es la manera más engorrosa de hacerlo. ¡Que tal si fueran 20, 30, 100 o 200 personas!.

Los arreglos (también llamados *arrays*, matrices o vectores) son espacios de memoria que se utilizan para almacenar información de **un tipo** de datos determinado. Podemos tener arreglos de enteros, de cadena, de reales de simple precisión, etc. Podemos imaginar un arreglo sencillo de la siguiente forma:



Cada cuadro (o ***dirección*** del arreglo) es un espacio reservado en memoria para guardar información del tipo que vaya a ser el arreglo. Siguiendo con el ejemplo de las 5 personas, se supone que el arreglo será de **tipo** String ya que almacenaremos cadenas de caracteres. Para utilizar un arreglo, debemos declararlo primero con la siguiente sintaxis:

DIM *nombre_del_arreglo***(**tamaño_del_arreglo**) AS** tipo_de_dato

Siguiendo con el ejemplo anterior, podemos declarar nuestro arreglo de la siguiente forma:

DIM personas(5)**) AS STRING**

Ahora veremos como llenar las localidades que reservamos en el arreglo.

```
' Rellena un arreglo con nombres de personas.

DIM personas(5) AS STRING  ' Declaramos nuestro arreglo sencillo

CLS
PRINT : COLOR 15
PRINT "CAPTURA DE PERSONAS:"
COLOR 7
INPUT "Persona 1 ->"; personas(1)
INPUT "Persona 2 ->"; personas(2)
INPUT "Persona 3 ->"; personas(3)
INPUT "Persona 4 ->"; personas(4)
INPUT "Persona 5 ->"; personas(5)

PRINT : COLOR 15
PRINT "SALIDA DE PERSONAS:"
COLOR 7
PRINT "Persona 1 ->"; personas(1)
PRINT "Persona 2 ->"; personas(2)
PRINT "Persona 3 ->"; personas(3)
PRINT "Persona 4 ->"; personas(4)
PRINT "Persona 5 ->"; personas(5)
END
```

En realidad el programa anterior es muy engorroso todavía, pero nos ilustra como es como se llenan las ***direcciones*** de un arreglo. De lo anterior podemos ver que el nombre de la persona 1 se almacenará en la dirección 1 del arreglo, la persona 2 en la dirección 2, la persona 3 en la dirección 3 del arreglo, etc. Si quisiéramos llenar una dirección 6 se produciría un mensaje "subscript out of range" ("límite fuera de rango"), ya que queremos poner valores en una dirección que no existe.

Como las direcciones están especificadas por un número, podemos utilizar un FOR para rellenar el arreglo en la posición del contador, empezando el contador en el límite inferior del arreglo y terminando en el límite superior. Veamos como se hace...

```
' Rellena un arreglo con nombres de personas. Forma óptima

DIM personas(5) AS STRING  ' Declaramos nuestro arreglo

CLS
' Entrada de datos
PRINT : COLOR 15
PRINT "CAPTURA DE PERSONAS:"
COLOR 7
FOR c = 1 TO 5
   PRINT "Persona "; c; "-> "; : INPUT "", personas(c) 'Rellena arreglo en la posición de c
NEXT c
```

```
' Salida de datos
PRINT : COLOR 15
PRINT "SALIDA DE PERSONAS:"
COLOR 7
FOR c = 1 TO 5
   PRINT "Persona "; c; "-> "; personas(c) ‘Imprime arreglo en la posición c
NEXT c
END
```

Existe otra forma de declarar un arreglo que es la siguiente:

DIM *nombre_del_arreglo***(**límite_inferior **TO** límite_superior**) AS** tipo_de_dato

De lo anterior podemos deducir que es posible tener arreglos cuyo límite inferior sea cualquier entero corto. Por ejemplo:

```
DIM elementos(3 TO 19) AS INTEGER
DIM goles(0 TO 15) AS INTEGER
DIM velocidades(-1000 TO 1000) AS SINGLE
```

El limite inferior por default es **0**. Como el límite inferior es cero, alguien se puede preguntar: bueno, si el límite inferior es cero, entonces ¿porqué empezamos a rellenar el arreglo desde la posición **1**?. Si consideramos el formato

DIM *nombre_del_arreglo***(**tamaño_del_arreglo**) AS** tipo_de_dato

podemos ampliar lo siguiente:

- Si empezamos a rellenar el arreglo con 1 como límite inferior (o ***base***), entonces QuickBASIC lo “entenderá” y nos dejará que lo rellenemos desde 1 para terminar en 5. En pocas palabras, si empezamos a rellenar en 1, el tamaño del arreglo permanece igual.

- Si queremos llenar el arreglo *personas(5)* (siguiendo con el ejemplo anterior) empezando desde 0, entonces terminaremos de introducir todas las personas en la posición 4 del arreglo. Pero como en realidad “tamaño_del_arreglo” es el límite superior, todavía podríamos introducir el nombre de una persona más en la posición 5 del arreglo. En pocas palabras si comenzamos a rellenar el arreglo desde 0, el tamaño del arreglo se incrementaría en 1.

Ya que QB lo permite, es recomendable empezar a rellenar el arreglo desde 1, aunque hay quién prefiere establecer desde el principio los límites del arreglo usando

```
DIM personas(1  TO 5) AS STRING
```

Aunque no es muy común que se utilicen arreglos cuyo límite inferior sea diferente de 1 (en BASIC), puede que se dé el caso y por eso se usa el TO para poner donde empieza y donde termina un arreglo.

ARREGLOS DE MÁS DE UNA DIMENSION

Hasta el momento hemos visto arreglos sencillos o de *una dimensión*, ahora veremos arreglos de más de una dimensión. Empecemos por ver un ejemplo con un array de dos dimensiones:

Supongamos que queremos capturar los siguientes datos de 6 personas para luego imprimirlos en el mismo orden en que entraron:

Nombre
Sexo
Clave Unica de Registro de Población (CURP)
Dirección
Zona postal

Nota: La zona postal puede verse a simple vista como para almacenarse tipo LONG o INTEGER, pero hay que tener en cuenta que sí tenemos un código postal que empiece en 0 (por ejemplo 03145), al guardarlo se almacenará sin el cero inicial (3145), lo cual no es cierto. ¡O que tal un código de barras de algún producto!, ¡una tarjeta de crédito!, ¡un número de expediente!, o ¡una cuenta bancaria!. Para evitar errores, estos datos deben entrar en formato STRING. Luego, ¿cuáles datos son los que deben entrar como número? Sencillo, solo los datos que nos sirvan para realizar **cálculos**.

A simple vista se puede antojar utilizar 6 (1 por cada persona) arreglos de 5 elementos (en cada dirección iría un dato) tipo String; o también 5 (1 por cada dato) arreglos de 6 elementos (en cada elemento iría una persona). Como esto es algo tedioso (imaginemos unas 100 personas y unos 30 datos para algún curriculum), lo mejor es utilizar un arreglo de **dos dimensiones**. En realidad la sentencia DIM no se limita a declarar variables o arreglos de 1 sola dimensión, sino que va mucho más allá:

```
DIM personas(1 TO 10, 1 TO 5) o DIM personas(10, 5)          'Arreglo de 2 dimensiones
DIM determinante(4, 4)                                       'Arreglo de 2 dimensiones
DIM abuelos(2, 3, 15) o DIM abuelos(1 TO 2, 1 TO 10, 1 TO 15)  'Arreglo de 3 dimensiones
DIM bisabuelos(2, 3, 2, 3)                                   'Arreglo de 4 dimensiones
```

Las **dimensiones** de un arreglo en QB pueden ser desde 0 (que es una variable sencilla) hasta 60 dimensiones; y el **rango de los límites** puede ser cualquier número en el intervalo de -36768 a 36767, pero el **número de elementos** no debe sobrepasar los 36767. Ahora veremos como utilizar un arreglo de dos dimensiones:

```
DIM personas(6, 5) AS STRING

'Entrada de datos
FOR individuo = 1 TO 6
  COLOR 10, 1
  CLS
  PRINT TAB(33); "CAPTURA DE DATOS": PRINT
  COLOR 15
  PRINT "DATOS DE LA PERSONA"; : COLOR 15: PRINT individuo
  COLOR 7: PRINT
  FOR dato = 1 TO 5
```

```
    IF dato = 1 THEN
      INPUT "Introduzca el nombre -->> ", personas(individuo, dato)
    ELSEIF dato = 2 THEN
      INPUT "Introduzca el sexo (M /F )->> ", personas(individuo, dato)
    ELSEIF dato = 3 THEN
      INPUT "Introduzca la CURP -->> ", personas(individuo, dato)
    ELSEIF dato = 4 THEN
      INPUT "Introduzca la dirección -->> ", personas(individuo, dato)
    ELSEIF dato = 5 THEN
      INPUT "Introduzca la zona postal -->> ", personas(individuo, dato)
    END IF
  NEXT dato
  PRINT
  COLOR 15
  PRINT TAB(22); "HECHO. PRESIONE <ENTER> PARA CONTINUAR"; : INPUT "", enter$
  CLS
NEXT individuo

'Salida de datos
FOR individuo = 1 TO 6
  COLOR 10, 1
  CLS
  PRINT TAB(33); "SALIDA DE DATOS": PRINT
  COLOR 15
  PRINT "DATOS DE LA PERSONA"; : COLOR 15: PRINT individuo
  COLOR 7: PRINT
  FOR dato = 1 TO 5
    IF dato = 1 THEN
      PRINT "Nombre -->> "; personas(individuo, dato)
    ELSEIF dato = 2 THEN
      PRINT "Sexo (M /F) -->> "; personas(individuo, dato)
    ELSEIF dato = 3 THEN
      PRINT "CURP -->> "; personas(individuo, dato)
    ELSEIF dato = 4 THEN
      PRINT "Dirección -->> "; personas(individuo, dato)
    ELSEIF dato = 5 THEN
      PRINT "Zona postal -->> "; personas(individuo, dato)
    END IF
  NEXT dato
  PRINT
  COLOR 15
  PRINT TAB(22); "HECHO. PRESIONE <ENTER> PARA CONTINUAR"; : INPUT "", enter$
  CLS
NEXT individuo
END
```

En el ejemplo anterior utilizamos un arreglo de dos dimensiones para almacenar los datos de las personas. La primera dimensión identifica a las personas en una dirección respectiva (la persona 1 se representa en la dirección 1 de la primera dimensión del arreglo, la persona 2 en la dirección 2, la persona 3 en la 3, etc.) y en la segunda dimensión se guardan los datos de cada persona.

Con el primer FOR nos posicionamos en la dirección de la persona y cuando con el segundo FOR terminamos de recolectar los datos, entonces se continua con la siguiente persona.

Los datos se guardan usando la conveniencia de que el elemento 1 de la segunda dimensión corresponde al nombre de la persona, el elemento 2 al sexo, el elemento 3 a la dirección, el elemento 4 a la CURP y el elemento 5 a la zona postal. Por esto puse los ELSEIF.

Ahora veamos como utilizar un arreglo de 3 dimensiones:

```
' Recolecta las horas trabajadas por semana de 5 trabajadores pertenecientes
' a 3 departamentos diferentes de 2 sucursales de una cadena comercial.

DIM empleados(2, 3, 5) AS INTEGER

FOR sucursal = 1 TO 2
  COLOR 15, 1
  CLS
  PRINT TAB(20); "CAPTURA DE HORAS TRABAJADAS SEMANALMENTE"
  PRINT
  COLOR 15
  PRINT TAB(30); "-- SUCURSAL"; sucursal; "--"
  PRINT
  FOR departamento = 1 TO 3
    COLOR 15
    PRINT "DEPARTAMENTO"; departamento
    FOR empleado = 1 TO 5
      COLOR 7
      PRINT "Horas trabajadas por el empleado"; empleado; "->> ";
      COLOR 10: INPUT "", empleados(sucursal, departamento, empleado)
    NEXT empleado
    PRINT
  NEXT departamento
NEXT sucursal
```

La primera dimensión del arreglo corresponde a la sucursal, la segunda parte al departamento en donde trabaja el empleado y en la 3ª dimensión se almacenará el no. de horas que trabajó en la semana.

El primer FOR nos sirve para primeramente colocarnos en la sucursal, el segundo FOR para colocarnos en el departamento, y el 3er. FOR nos sirve para recolectar las horas trabajadas de los empleados; cuando este ultimo FOR termina, entonces se prosigue con el siguiente departamento; y cuando se terminan los departamentos, entonces continuamos con la siguiente sucursal.

FUNCIONES LBOUND Y UBOUND

Estas dos funciones nos permiten obtener los límites inferior y superior de la dimensión de un arreglo (UBOUND, abreviatura de *Upper Bound* o límite superior; LBOUND, abreviatura de *Lower Bound* o límite inferior) respectivamente. Su sintaxis:

UBOUND(*nombre_del_arreglo*[,dimensión])
LBOUND(*nombre_del_arreglo*[,dimensión])

Si el arreglo es de una dimensión, entonces solo ponemos el nombre del arreglo. Por ejemplo si tenemos el arreglo

```
DIM cubo(1 TO 20, 3 TO 9, 3 TO 15) AS SINGLE
```

entonces

```
LBOUND(cubo, 1)     'Devolverá 1 que es el límite inferior de la primera dimensión
LBOUND(cubo, 2)     'Devolverá 3 que es el límite inferior de la segunda dimensión
LBOUND(cubo, 3)     'Devolverá 3 que es el límite inferior de la tercera dimensión
```

y

```
UBOUND(cubo, 1)     'Devolverá 20 que es el límite superior de la primera dimensión
UBOUND(cubo, 2)     'Devolverá 9 que es el límite superior de la segunda dimensión
UBOUND(cubo, 3)     'Devolverá 15 que es el límite superior de la tercera dimensión
```

¿Que pasaría si quisiéramos obtener los límites superior e inferior del siguiente arreglo (ojo que no especificamos los limites con TO)?

```
DIM vectores(3) AS INTEGER
```

Es posible que si en un programa empezamos a rellenar este arreglo desde 1 creamos que como empezamos a rellenar desde 1, entonces LBOUND nos retornará 1; lo cual no hay nada más falso. Por otro lado y como los arreglos comienzan virtualmente en 0, entonces:

```
LBOUND(vector)      'Devolverá 0 ya que los arreglos empiezan desde 0.
UBOUND(vector)      'Devolverá 3 ya que en efecto el límite superior es 3.
```

De lo anterior podemos deducir que:

- ➔ Suponiendo que empezamos a rellenar un arreglo desde un número que no sea cero y luego queremos referirnos al menor elemento en una determinada dimensión utilizando LBOUND, entonces deberemos poner los limites usando TO. Ejemplo:

```
DIM prueba(3 TO 6)
CLS
PRINT "El limite inferior del arreglo prueba es ", LBOUND(prueba)
PRINT "El limite superior del arreglo prueba es ", UBOUND(prueba)
```

- ➔ Podemos utilizar la declaración **OPTION BASE n** para indicar que el limite inferior de todos los arreglos de nuestro programa que no declaremos con TO empezarán en *n*, donde *n* puede ser **0** o **1**. Ejemplo:

```
OPTION BASE 0
DIM prueba(3 TO 6)
DIM perros(3)
```

```
CLS
PRINT "El limite inferior del arreglo prueba es ", LBOUND(prueba)
PRINT "El limite superior del arreglo prueba es ", UBOUND(prueba)
PRINT
PRINT "El limite inferior del arreglo perros es ", LBOUND(perros)
PRINT "El limite superior del arreglo perros es ", UBOUND(perros)
```

Notas:

- Si se va a utilizar la declaración OPTION BASE, entonces esta debe de ir antes de cualquier arreglo.

ARREGLOS ESTÁTICOS Y DINÁMICOS

Existen dos tipos de arreglos: **estáticos** y **dinámicos**. Una arreglo **estático** es un arreglo cuyo número de elementos en cada dimensión permanecerá **igual** durante le ejecución del programa; por otro lado un arreglo **dinámico** es un arreglo cuyo número de elementos en cada dimensión puede **cambiar** en el transcurso del programa; o sea, puede hacerse más chico o más grande. Hasta el momento solo hemos manejado arreglos estáticos. Ya que los arreglos ocupan espacio en memoria, debemos indicarle al compilador si queremos que "haga flexible" a los arreglos y a la memoria, ó indicarle que los arreglos permanecerán con el mismo número de elementos durante todo el programa.

Si vamos a usar arreglos dinámicos en nuestro programa, entonces debemos de poner el metacomando (comando a nivel compilador) **$DYNAMIC** al inicio del programa y a manera de comentario, esto es:

```
' $DYNAMIC
    o
REM $DYNAMIC
```

Por otro lado si en nuestro programa solo vamos utilizar arreglos estáticos, entonces podemos poner el metacomando **$STATIC** al inicio del programa, aunque no es necesario.

```
'$STATIC
   o
REM $STATIC
```

Nota: Todos los arreglos por default son estáticos y tipo SINGLE.

ERASE Y REDIM

La sentencia **ERASE** reinicializa un arreglo poniendo todos los elementos a cero y todas las cadenas en nulo ("") (en pocas palabras lo "resetea"). Ejemplo:

```
' Ejemplo de borrado de arreglos.

DIM prueba(1 TO 5)
```

```
CLS
PRINT "Le damos valores al arreglo 'prueba':"
FOR i = 1 TO 5 STEP 1
   prueba(i) = i
   PRINT "Prueba ("; i; ") --> "; prueba(i)
NEXT i
PRINT

ERASE prueba
PRINT "Luego de resetear el arreglo 'prueba':"
FOR i = 1 TO 5 STEP 1
   PRINT "Prueba ("; i; ") --> "; prueba(i)
NEXT i
```

Para borrar más de un arreglo solo hay que ponerlos separados por comas (,) después de ERASE. Por otro lado, la sentencia **REDIM** cambia el numero de elementos que contienen las dimensiones de un arreglo $DYNAMIC.

```
' Ejemplo de arreglo dinámicos.

REM $DYNAMIC

DIM prueba(1 TO 5)

CLS
PRINT "Al principio el limite inferior del arreglo es "; LBOUND(prueba)
PRINT "y el límite superior es "; UBOUND(prueba)
PRINT

REDIM prueba(7 TO 200)          'Cambiamos lo límites del arreglo.
PRINT "Luego el limite inferior del arreglo es "; LBOUND(prueba)
PRINT "y el límite superior es "; UBOUND(prueba)
END
```

Nota: REDIM cambia el **número de elementos** en las dimensiones de un arreglo, **NO** las dimensiones; o sea, NO podemos tener algo como.

```
REM $DYNAMIC
DIM arreglo(5, 6)              ' Array de 2 dimensiones
REDIM arreglo(3, 5, 6)         ' Array de  3 dimensiones
```

Por último, solo queda mencionar que al cambiar el tamaño del un arreglo, todos los elementos se resetean (números a 0 y cadenas a "”). También es posible cambiar el número de elementos del arreglo $DYNAMIC, primero borrándolo con ERASE y luego volver a declarar los límites con DIM.

```
' Ejemplo de arreglo dinámicos II.

REM $DYNAMIC
```

```
DIM prueba(1 TO 5)

CLS
PRINT "Al principio el límite inferior del arreglo es "; LBOUND(prueba)
PRINT "y el límite superior es "; UBOUND(prueba)
PRINT

ERASE prueba                    'Primero lo borramos...
DIM prueba(5 TO 20)             ' y luego lo volvemos a declarar.

PRINT "Luego el limite inferior del arreglo es "; LBOUND(prueba)
PRINT "y el límite superior es "; UBOUND(prueba)
```

Capítulo 5: Programación estructurada

Al parecer, esta forma de programar se basa fundamentalmente en la inclusión en un programa de **subprogramas** que hagan cada uno una tarea especial para el programa en general. Como *"la necesidad es la madre de la invención"*, al parecer la programación estructurada nace de la necesidad de dotar a los programas de una estructura, una mayor claridad en el diseño y una especie de jerarquización que permitan depurar, mantener y corregirlos de una manera sencilla y rápida.

QuickBASIC puede incluir en un programa:

- ➔ Subrutinas
- ➔ Procedimientos
- ➔ Funciones
- ➔ Módulos

SUBRUTINAS

Las subrutinas son **subprogramas** que son llamados desde un programa principal u otra subrutina mediante la orden **GOSUB**. Inician con una etiqueta o número de línea y terminan con la sentencia **RETURN** ("regreso"), la cual le devuelve el control al nivel inmediato desde donde fue llamada (ya sea programa principal u otra subrutina). Vamos a poner como ejemplo uno de los programas de los capítulos anteriores, pero ahora al estilo estructurado:

```
' Ejemplo de subrutinas GOSUB (gosub1.bas)

DO
  CLS
  COLOR 10
  PRINT TAB(29); "MENÚ DE OPERACIONES"
  PRINT TAB(27); "-----------------------"
  COLOR 7
  PRINT TAB(32); "1) SUMAR"
  PRINT TAB(32); "2) RESTAR"
  PRINT TAB(32); "3) MULTIPLICAR"
  PRINT TAB(32); "4) DIVIDIR"
  PRINT TAB(32); "5) EXPONENCIAR"
  PRINT TAB(32); "6) RAIZ CÚBICA APROXIMADA"
  PRINT TAB(32); "7) SALIR"

  PRINT : PRINT
  PRINT TAB(30); : INPUT "ESCOGA SU OPCIÓN -> ", opcion
  CLS

  SELECT CASE opcion
    CASE IS = 1
      GOSUB sumar          ' Nos manda a la subrutina "sumar"
    CASE IS = 2
      GOSUB restar         ' Nos manda a la subrutina "restar"
    CASE IS = 3
      GOSUB multiplicar    ' Nos manda a la subrutina "multiplicar"
    CASE IS = 4
      GOSUB dividir        ' Nos manda a la subrutina "dividir"
    CASE IS = 5
      GOSUB exponenciar    ' Nos manda a la subrutina "exponenciar"
```

```
    CASE IS = 6
      GOSUB radicar        ' Nos manda a la subrutina "radicar"
    CASE IS = 7
      EXIT DO              ' Sale del bucle DO
    CASE ELSE
      COLOR 18             ' Color verde intermitente
      PRINT TAB(30); "--OPERACIÓN NO VALIDA--"
  END SELECT
  COLOR 7: PRINT
  INPUT "PRESIONA <ENTER> PARA REGRESAR", enter$
LOOP
END

sumar:          'Inicia la subrutina sumar...
  COLOR 10
  PRINT TAB(38); "SUMAR"
  PRINT TAB(36); "---------"
  PRINT : COLOR 7
  INPUT "Primer número -> ", n1
  INPUT "Segundo número -> ", n2
  PRINT "La suma de"; n1; "y"; n2; "es"; n1 + n2
RETURN          'Regresa el control al nivel desde donde fue llamada
restar:         'Inicia la subrutina restar...
  COLOR 10
  PRINT TAB(37); "RESTAR"
  PRINT TAB(36); "--------"
  PRINT : COLOR 7
  INPUT "Primer número -> ", n1
  INPUT "Segundo número -> ", n2
  PRINT "La resta de"; n1; "menos"; n2; "es"; n1 - n2
RETURN          'Regresa el control al nivel desde donde fue llamada
multiplicar:
  COLOR 10
  PRINT TAB(34); "MULTIPLICAR"
  PRINT TAB(32); "---------------"
  PRINT : COLOR 7
  INPUT "Primer número -> ", n1
  INPUT "Segundo número -> ", n2
  PRINT "El producto de"; n1; "por"; n2; "es"; n1 * n2
RETURN
dividir:
  COLOR 10
  PRINT TAB(36); "DIVIDIR"
  PRINT TAB(35); "---------"
  PRINT : COLOR 7
  INPUT "Dividendo -> ", n1
  INPUT "Divisor -> ", n2
  PRINT "El cociente de"; n1; "entre"; n2; "es"; n1 / n2
RETURN
exponenciar:
  COLOR 10
  PRINT TAB(34); "EXPONENCIAR"
  PRINT TAB(33); "-------------"
  PRINT : COLOR 7
  INPUT "Base -> ", n1
  INPUT "Potencia -> ", n2
```

```
  PRINT "El resultado de elevar"; n1; "a la"; n2; "es"; n1 ^ n2
RETURN
radicar:
  COLOR 10
  PRINT TAB(34); "RAIZ cúbica aproximada”
  PRINT TAB(33); "-------------"
  PRINT : COLOR 7
  INPUT "Número para sacarle raíz cúbica aproximada -> ", n1
  DO WHILE (cont * cont * cont) < n1
    cont = cont + .001
  LOOP
  PRINT "La raíz cúbica de"; n1; "es” ; cont
RETURN
```

Las subrutinas del tipo GOSUB...RETURN tienen la siguiente estructura:

Sintaxis:

etiqueta o número de línea:
  <sentencias>
RETURN [etiqueta o número de línea]

Donde RETURN devolverá el control al nivel desde donde fue llamado…

```
' Demuestra como RETURN devuelve el control al nivel desde donde la
' subrutina fue llamada (gosubbak.bas)

CLS
PRINT "-> Estamos en el programa principal..."
INPUT "Quieres ir a la subrutina 1 (s/n) -> ", ir$
IF UCASE$(ir$) = "S" THEN GOSUB Subrutina1       ' Vamos a la subrutina 1
PRINT "-> estamos en el programa principal y ya acabamos."
END

Subrutina1:
  PRINT
  PRINT "-> Estas en la subrutina 1"
  INPUT "Quieres regresar a principal (s/n) -> ", reg$
  IF UCASE$(reg$) = "N" THEN GOSUB Subrutina2   ' Vamos a la subrutina 2
  PRINT "-> vas a regresar a principal desde la subrutina 1"
  RETURN     ' A principal, desde donde esta subrutina fue llamada

Subrutina2:
  PRINT
  PRINT "-> Estamos en la subrutina 2"
  PRINT
  RETURN     ' A subrutina 1, desde donde esta subrutina fue llamada
```

o a otra etiqueta o numero de linea…

```
' Ejemplo de GOSUB que retorna el control a otra etiqueta, pero dentro del nivel desde
```

```
' donde fue llamada.

CLS
PRINT "-> en el programa principal"
GOSUB Hola
Adios:
   PRINT "-> regresamos al programa principal"
   PRINT "Adiós"
   END

Hola:
   PRINT "Hola a todos..."
   PRINT "esta subrutina fue llamada desde principal"
   GOSUB HastaPronto
   PRINT "esta línea en subrutina 1 puede ser omitida"
Salida:
   PRINT "-> de regreso en subrutina uno"
RETURN Adios

HastaPronto:
   PRINT "-> en subrutina dos"
   PRINT "Hasta pronto..."
RETURN Salida
```

COMO USAR GOSUB Y RETURN

Aunque podemos usar RETURN para retornar el control a otro subprograma diferente del que llamó a esa subrutina, para evitar conflictos y tener una mejor estructura es recomendable usarlo para devolver el control al nivel inmediato desde donde la subrutina fue llamada, y por otro lado utilizar a GOSUB para darle el control a otra subrutina cualquiera. Lo anterior nos evita tener errores como el siguiente:

```
CLS
PRINT "-> Estamos en el programa principal"
GOSUB Saludo
END

Saludo:
   PRINT "-> Estamos en subrutina Saludo"
   PRINT "hola a todos"
   RETURN SaludoII   'Utilizamos RETURN en vez de GOSUB para mandar
                     'el control a otra subrutina
SaludoII:
   PRINT "-> Estamos en subrutina SaludoII"
   RETURN            'Producirá un error ya que este RETURN no devuelve
                     'el control a un GOSUB respectivo
```

Estas subrutinas de tipo GOSUB...RETURN se deben escribir después del final del programa (END) para evitar que el programa se vaya de largo en la ejecución y nos produzca un error como el siguiente:

```
CLS
```

```
PRINT "-> Estamos en el programa principal"
GOSUB Saludo
' END debería de ir aquí para evitar que se vaya de largo y encuentre un
' RETURN sin GOSUB.

Saludo:
  PRINT "-> Estamos en subrutina saludo"
  PRINT "hola a todos"
RETURN
```

El siguiente tipo de **procedimientos** provee una manera más potente, sólida, sexy y legible que las subrutinas GOSUB...RETURN...

PROCEDIMIENTOS

Los procedimientos son subprogramas que reciben variables (o constantes) desde el nivel superior para ejecutar una determinada tarea. Veamos el siguiente ejemplo:

```
' Ejemplo de SUB #1
CONST saludo$ = "HOLA A TODOS..."

CLS
CALL Display(saludo$)
END

SUB Display (mensaje AS STRING)
  PRINT mensaje
END SUB
```

El ejemplo anterior nos presenta un simple programita que llama o invoca (mediante la orden **CALL**) a un procedimiento llamado *Display* para pasarle una constante String que contiene la cadena "*HOLA A TODOS...*" e imprimirla en pantalla. Ahora entremos en detalles:

Un procedimiento tiene la siguiente estructura:

Sintaxis:

SUB *nombre_del_procedimiento***[**(parámetros)**]** **[**STATIC**]**
    <instrucciones>
    [EXIT SUB**]**
    <instrucciones>
END SUB

Del **nombre** del procedimiento debemos decir que este tiene que ser **único** en todo el programa, es decir, NINGUN otro procedimiento ni variable debe de llamarse igual a este. Si en un programa existe algo como:

```
False = 0                    'False igual a cero
True = NOT False             'True diferente de cero
CLS
```

```
INPUT "¿Imprimimos mensaje (si = 1, no = 0)"; imprimir' Variable que se llama Imprimir
IF imprimir = True THEN CALL Imprimir
END

SUB Imprimir    ' ¡¡¡Procedimiento que se llama igual que una variable!!!
   PRINT "Hola"
END SUB
```

QuickBASIC nos presentará el mensaje "*Duplicate definition*", que significa que existe ya una variable con ese nombre.

MANEJO DE PARAMETROS Y ARGUMENTOS

Ahora, siguiendo con el ejemplo

```
' Ejemplo de SUB #1
CONST saludo$ = "HOLA A TODOS..."

CLS
CALL Display(saludo$)
END

SUB Display (mensaje AS STRING)
   PRINT mensaje
END SUB
```

alguien puede pensar lo siguiente: ¿Si al procedimiento le "pasamos" una constante llamada *saludo$*, entonces porqué dentro de la SUB este mismo valor se llama *mensaje*?. El **parámetro** *mensaje* nos indica que le "pasaremos" al procedimiento un **argumento** de tipo String, y que ese mismo argumento o valor (en este caso es una constante llamada *saludo$)* se llamará diferente dentro de la SUB (en este caso *mensaje*, ¿entiendes?). Veamos este otro ejemplo:

```
'Ejemplo #2 de SUB

CLS
INPUT "Introduce dos números  -->> ", a, b
CALL ImprimeSuma(a, b)   'Llamamos a la SUB ImprimeSuma.
END

SUB ImprimeSuma(n1, n2)
   PRINT "La suma de"; n1; "y"; n2; "es"; n1 + n2
END SUB
```

Aquí tenemos un procedimiento que se llama *ImprimeSuma* con dos parámetros (Single por default) *n1* y *n2*, al cual le pasaremos desde el programa principal dos variables (los argumentos) llamadas *a* y *b* para que imprima la suma de ambas.

VALORES POR REFERENCIA

Es muy importante dejar muy en claro que las variables se pasan a los procedimientos por **referencia**; es decir, se pasan con otro nombre, pero sigue siendo la **misma dirección en**

memoria; por lo tanto, todo lo que se les haga dentro del procedimiento las afectará afuera de este:

```
' Ejemplo #3 de SUB. Ejemplo de valores por referencia.
' Este programa no hace más que cambiar los valores de dos variables
' dentro de un procedimiento, para luego imprimirlos fuera de este.

CLS
INPUT "Introduce un numero  -->> ", a
INPUT "Introduce otro numero  -->> ", b
PRINT
COLOR 15: PRINT "Primero:"
COLOR 7
PRINT "a = "; a; " y b = "; b
CALL Cambiar(a, b)   'Llama al procedimiento cambiar.
COLOR 15: PRINT "Luego:"
COLOR 7
PRINT "a = "; a; " y b = "; b
END

SUB Cambiar (x, y)
   SWAP x, y         'Intercambia los valores.
END SUB              'Listo, vámonos.
```

En el ejemplo anterior intercambiamos los valores de dos variables (llamadas *a* y *b* en el programa principal) dentro de un procedimiento llamado *Cambiar* (en el que las variables se llaman *x* y *y*). Esto consiste una gran ventaja para cuando queremos obtener más de un valor en una misma serie de operaciones, lo cual no lo permiten las funciones.

PASANDO ARREGLOS

Ampliando un poco más esto, si queremos que los argumentos sean arreglos, estos se pasan de la siguiente manera:

```
DECLARE SUB RellenaArreglo (array() AS INTEGER)
DECLARE SUB ImprimeArreglo (array() AS INTEGER)
' Ejemplo de SUB #4. Arreglos como parámetros

'Arreglo entero estático de dos dimensiones
DIM arreglo(1 TO 2, 1 TO 2) AS INTEGER

CLS
CALL RellenaArreglo(arreglo())
CALL ImprimeArreglo(arreglo())
END

SUB ImprimeArreglo (array() AS INTEGER)
   FOR i = LBOUND(array, 1) TO UBOUND(array, 1)    'Limites de la 1ª dimensión
      FOR j = LBOUND(array, 2) TO UBOUND(array, 2) 'Limites de la 2ª dimensión
         PRINT "Elemento"; i; j; "-->> "; array(i, j)
      NEXT j
   NEXT i
END SUB
```

```
SUB RellenaArreglo (array() AS INTEGER)
  FOR i = LBOUND(array, 1) TO UBOUND(array, 1)    'Limites de la 1ª dimensión
    FOR j = LBOUND(array, 2) TO UBOUND(array, 2) 'Limites de la 2ª dimensión
      array(i, j) = i * j  'Rellena el arreglo (nada importante)
    NEXT j
  NEXT i
END SUB
```

No tiene ninguna ciencia, solo se pasa el arreglo **sin nada entre paréntesis**, a menos que queramos pasar una posición determinada del arreglo en particular.

Nota: Ojo que el parámetro también debe ser un arreglo **del mismo tipo**, de otra forma obtendremos un error

que significa que el tipo del parámetro no coincide con el de la variable que queremos pasar (argumento).

Pero bueno, para aquellos que ya detectaron algo diferente en el programa anterior, vamos a comentarlo. ¿Qué es eso de **DECLARE** SUB?. Si estas trabajando sobre QuickBASIC de seguro ya te diste cuenta que cada vez que guardas un programa en el que hay SUB's, QB **declara** por nosotros los procedimientos que estamos utilizando mediante la sentencia DECLARE. La sentencia DECLARE provoca que el compilador cheque el numero y tipo de los variables que le pasamos al procedimiento cuando lo mandamos llamar.

Sintaxis:
DECLARE {SUB | FUNCTION} *nombre* ([lista_de_parametros])

Nota: Si el tipo de los argumentos que pasamos no coincide con el tipo de los parámetros, entonces se produce el error:

que significa que el tipo del parámetro no coincide con el de la variable que queremos pasar.

FUNCIONES

Como ya vimos en el capítulo 4, una función es una correspondencia en la que, mediante operaciones con una o más variables independientes, le damos un valor a una variable dependiente. Hasta el momento hemos visto solamente las funciones predefinidas que nos ofrece QB, pero ahora vamos a ver como crear nuestras propias funciones de acuerdo a como las vayamos necesitando. Una función esta compuesta por un bloque de la siguiente forma:

Sintaxis:

FUNCTION nombre**[**identificador**]** (**[**parámetro AS tipo**[**, parámetro AS tipo**[**,...**]]]**) **[**STATIC**]**
  <sentencias>
  nombre = expresión
  [EXIT FUNCTION**]**
  <sentencias>
END FUNCTION

Veamos el siguiente ejemplo:

```
' Ejemplo de funciones #1. Función que devuelve el cuadrado de un número
DECLARE FUNCTION Cuad! (n AS SINGLE)

CLS
INPUT "Número ->> ", numero
cuadrado = Cuad(numero)        'Usando nuestra función
PRINT "El cuadrado es"; cuadrado
END

FUNCTION Cuad! (n AS SINGLE)
  Cuad = n * n  'Le asignamos el valor que devolverá
END FUNCTION
```

Dentro del cuerpo de la función, debemos de asignarle a la función el valor que va a devolver, en este caso a la función *Cuad* le asignamos a *n * n* para que lo devuelva a la variable dependiente *cuadrado*. Veamos otro ejemplo:

```
DECLARE FUNCTION Hip! (a!, b!)

CLS
COLOR 10: PRINT TAB(30); "CALCULO DE LA HIPOTENUSA"
COLOR 7: PRINT
INPUT "Cateto opuesto -> ", cop!
INPUT "Cateto adyacente -> ", cad!
hipotenusa! = Hip(cop!, cad!)
PRINT "La hipotenusa vale "; hipotenusa!

FUNCTION Hip! (a!, b!)
  c! = SQR(a! * a! + b! * b!)   'Teorema de pitágoras
  Hip! = c!                     'Le damos a Hip el valor de c!
END FUNCTION
```

PASANDO VALORES POR VALOR

El pasar argumentos por valor, nos permite pasar al subprograma el valor de las variables y no la dirección; esto evita que puedan ser modificadas dentro del subprograma. Para hacer esto, al llamar a la función o sub debemos encerrar al argumento entre paréntesis, por ejemplo de la siguiente forma:

DECLARE FUNCTION Presion(fuerza, area)
...
result = Presion(**(F)**, **(A)**)

FUNCIONES RECURSIVAS

Una función, al igual que una SUB, tiene la ventaja de que **puede llamarse a si misma**. Para visualizar esto mejor, pensemos en algo como calcular el resultado de elevar un número a una potencia. En este caso podemos decir que, por ejemplo, 8 elevado a la 3 sería

$$8^3 = 8 * 8 * 8 = 512$$

o también

$$8^3 = 8 * (8^2) = 8 * 8 * 8 = 512$$

Y para números cualquiera podemos tener que

$$x^y = x * x^{(y-1)} = x * x * x^{(y-2)} = x * x * x * x^{(y-3)} = x * x * x * ... * x^1$$

Ahora, un programa que hace lo mismo es:

```
'Utiliza una función recursiva para elevar un numero a otro.
DECLARE FUNCTION Pow (x, y)

CLS
INPUT "Introduce la base -> ", basse   'basse y no base
INPUT "Introduce la potencia -> ", potencia
resultado = Pow(basse, potencia)
PRINT basse; "a la "; potencia; "da "; resultado
END

FUNCTION Pow (x, y)
  IF y = 0 THEN
     Pow = 1       'Debemos evitar que se siga llamando
     EXIT FUNCTION
  ELSE
     Pow = x * Pow(x, y - 1)
  END IF
END FUNCTION
```

En el ejemplo anterior hemos jugado un poco con la recursividad y tenemos una función que se llama a si misma. Es **importante** que demos una **salida** a la recursión para que se vayan retornando los valores al nivel superior inmediato.

El factorial de un número es, por ejemplo, de 5:

5! = 5 * 4 * 3 * 2 * 1 = 120

que sería lo mismo que

5! = 5 * 4! = 5 * 4 * 3! = 120

```
'Calcula el factorial de un número usando una función que se llama a si misma

DECLARE FUNCTION Factorial (n AS INTEGER)

DIM numero AS INTEGER
CLS
INPUT "Introduzca el número para sacar su factorial -> ", numero
PRINT Factorial(numero)
END

FUNCTION Factorial (n AS INTEGER)
  IF n = 0 THEN
    Factorial = 1  'Por definición. Aquí se para la recursión
  ELSE
    Factorial = n * Factorial(n - 1)
  END IF
END FUNCTION
```

FUNCIONES Y PROCEDIMIENTOS ESTÁTICOS

Cada vez que nosotros accedemos a una función ó SUB, los valores de las variables que se encuentran en el cuerpo de esta se resetean (numero a 0 y cadenas a ""). Si queremos que los valores de estas se conserven entre llamadas, podemos hacer que la función (ó SUB) sea **estática** colocando la palabra **STATIC** al final de esta.

Ejemplo:

```
'Ejemplo de función STATIC
DECLARE FUNCTION Contador ()

CLS
DO
  c = Contador
  PRINT "Contador vale"; c
LOOP UNTIL c = 5
END

FUNCTION Contador STATIC   'Función estática sin parámetros
  i = i + 1
  Contador = i
END FUNCTION
```

De otra manera, si **no** fuera estática, tendríamos lo siguiente:

```
DECLARE FUNCTION Contador ()

CLS
DO
   c = Contador
   PRINT "Contador vale"; c
   INPUT "Desea salir (s/n) ", salir$                    'Proporcionamos una salida alternativa
   IF UCASE$(salir$) = "S" THEN EXIT DO
LOOP UNTIL c = 5
END

FUNCTION Contador    'Función dinámica sin parámetros.
   i = i + 1
   Contador = i
END FUNCTION
```

ALCANCE DE LAS VARIABLES

Las variables se pueden dividir por el alcance que tienen en dos tipos: **globales** y **locales**.

Variables locales

Las variables **locales** se crean dentro de cada procedimiento y su valor es únicamente para ese procedimiento. En el ejemplo

```
'Ejemplo de variables locales #1.
DECLARE SUB ejemplo (n AS INTEGER)

DIM num AS INTEGER
num = 19

CLS
PRINT "-> Estamos en el programa principal"
PRINT "num vale"; num
PRINT "n vale"; n
PRINT
CALL ejemplo(num)
END

SUB ejemplo (n AS INTEGER)
   PRINT "-> Estamos dentro del procedimiento"
   PRINT "num vale"; num
   PRINT "n vale"; n; "(valor de num)"
END SUB
```

tenemos dos variables, *num* y *n*. Ambas variables son locales, pero *num* es local a principal y *n* al procedimiento *ejemplo*. Esto quiere decir que si citamos a *num* dentro del procedimiento, entonces se crea una nueva variable local (pero ahora local al procedimiento) y aunque fuera de ese procedimiento ya existe una variable con ese mismo nombre, ambas son diferentes

¿entiendes?. Todas las variables son locales por default, tienen alcance únicamente en sus respectivos procedimientos.

Variables globales

Las variables globales son aquellas cuyo valor será el mismo en todos los procedimientos, sin necesidad de pasarlas como argumentos. Debemos utilizar la orden **SHARED** ("compartido") para indicar que un mismo nombre de variable tendrá el mismo valor en todos los procedimientos de **ese módulo**. Si la variable es creada **dentro de una SUB** entonces se usa la sintaxis

Sintaxis:

SHARED variable [AS tipo] [, variable [AS tipo]]...

```
'Ejemplo de variables globales #1.
'La variable global se crea dentro de un procedimiento...
DECLARE SUB Ejemplo ()

CLS
CALL Ejemplo   'Invocamos nuestra sub
PRINT "En principal e también vale"; e; ":-)"
END

SUB Ejemplo
  SHARED e   'Variable compartida en todos los procedimientos de este modulo
  e = 2.718282
  PRINT "En el procedimiento e vale"; e
END SUB
```

De otra forma utilizaremos SHARED después de DIM.

Sintaxis:

DIM SHARED variable [AS tipo] [, variable [AS tipo]]...

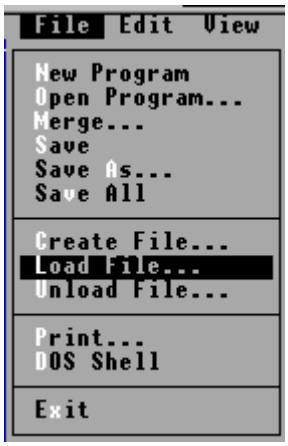
```
'Ejemplo de variables globales #2.
' La variable se crea fuera de un procedimiento...
DECLARE SUB Ejemplo ()

CLS
DIM SHARED pi, e
pi = 3.1416: e = 2.718282
PRINT "En principal e vale"; e; "y pi"; pi
CALL Ejemplo

SUB Ejemplo
  PRINT "En el procedimiento e vale"; e; "y pi"; pi; "también."
END SUB
```

MÓDULOS

Un programa BASIC esta compuesto por uno o más módulos (.bas). Hasta ahora, solo hemos manejado programas de un solo módulo llamado **módulo principal**. El módulo principal es el "kernel del programa", es donde deben entrar los datos e iniciar el programa. Los otros módulos pueden contener SUBs, funciones, constantes, y tipos de datos definidos por nosotros (luego los veremos). La creación de módulos constituye un método muy potente para reutilizar código, ya que podemos utilizarlos para realizar otros programas.

Cuando nosotros creamos un módulo nuevo, QB lo guardara en el disco duro (o dispositivo que queramos) con el nombre que le especifiquemos y una extensión .bas. Al compilar un programa de dos o más módulos, QB compilará separadamente cada modulo para luego enlazar los .obj y formar el ejecutable.

Nota: En QuickBASIC puedes crear un nuevo módulo mediante el menú **File/Create File...**; si existe, cargarlo usando **File/Load File...**, y descargarlo con **File/Unload File...**,



así como editar o mover sus subprogramas con **<F2>**. Veamos como podría ser la estructura de un módulo:

```
'Módulo modStad.bas.
'Funciones estadísticas para obtener la media, mediana y moda de una
'muestra.
DECLARE SUB Bubble (arreglo() AS SINGLE)
DECLARE FUNCTION GetModa (muestra() AS SINGLE)
DECLARE FUNCTION GetMediana (muestra() AS SINGLE)
DECLARE FUNCTION GetMedia (muestra() AS SINGLE)

' Esta sub ordenara los elementos del arreglo de menor a mayor mediante el
' el método de la burbuja.
SUB Bubble (arreglo() AS SINGLE)
   liminf = LBOUND(arreglo)
   limsup = UBOUND(arreglo)
    FOR i = liminf TO limsup - 1
      FOR j = liminf TO limsup - 1
         IF arreglo(j) > arreglo(j + 1) THEN SWAP arreglo(j), arreglo(j + 1)
      NEXT j
```

1.- Sección de declaraciones

```
  NEXT i
END SUB

FUNCTION GetMedia (muestra() AS SINGLE)
  liminf = LBOUND(muestra)
  limsup = UBOUND(muestra)
  suma = 0
  FOR i = liminf TO limsup
    suma = suma + muestra(i)
    c = c + 1
  NEXT i
  GetMedia = suma / c
END FUNCTION

FUNCTION GetMediana (muestra() AS SINGLE)
  liminf = LBOUND(muestra)
  limsup = UBOUND(muestra)
  GetMediana = (muestra(liminf) + muestra(limsup)) / 2
END FUNCTION

'Esta función retornara solo UNA moda, y se debe pasar el arreglo ORDENADO
'de menor a mayor. Puedes modificarla para mejorarla.
FUNCTION GetModa (muestra() AS SINGLE)
  liminf = LBOUND(muestra)
  limsup = UBOUND(muestra)
  DIM contador(liminf TO limsup)

  FOR i = liminf TO limsup
    FOR j = liminf TO limsup
      IF muestra(i) = muestra(j) THEN contador(i) = contador(i) + 1
    NEXT j
  NEXT i

  max = 1
  FOR i = liminf TO limsup
    IF contador(i) > max THEN max = i
  NEXT i
  GetModa = muestra(max)
END FUNCTION
```

Como ya vimos anteriormente, cada vez que guardamos un módulo que contiene subprogramas QB automáticamente los declara al inicio, pero para poder utilizar estos mismos subprogramas en otro módulo, hay que declararlos también en este. El siguiente ejemplo muestra un modulo principal que utiliza al modulo anterior:

```
DECLARE SUB Bubble (arreglo() AS SINGLE)
DECLARE FUNCTION GetMedia (arreglo() AS SINGLE)
DECLARE FUNCTION GetMediana (arreglo() AS SINGLE)
DECLARE FUNCTION GetModa (arreglo() AS SINGLE)

DIM estaturas(1 TO 5) AS SINGLE

CLS
PRINT TAB(15); "CALCULA LA MEDIA, MEDIANA Y MODA DE 5 ESTATURAS"
```

```
PRINT : PRINT
FOR i = 1 TO 5
   PRINT "Estatura"; i; "-> "; : COLOR 15: INPUT "", estaturas(i)
   COLOR 7
NEXT i
PRINT

CALL Bubble(estaturas())              'Esta sub ordenará  el arreglo
media = GetMedia(estaturas())
mediana = GetMediana(estaturas())
moda = GetModa(estaturas())

COLOR 7: PRINT "La media es "; : COLOR 15: PRINT media
COLOR 7: PRINT "La mediana  es "; : COLOR 15: PRINT mediana
COLOR 7: PRINT "La moda  es "; : COLOR 15: PRINT moda
END
```

VARIABLES ENTRE MÓDULOS

Para poder compartir las variables entre los procedimientos de otros módulos, debemos utilizar la palabra **COMMON** ("común") antes de SHARED y luego el nombre de la(s) variable(s) que queremos que sean comunes; esto en **todos** los módulos que compartirán las variables.

Sintaxis:

COMMON SHARED variable [AS tipo] [, variable [AS tipo]]...

```
'Modulo principal common.bas
DECLARE FUNCTION Cuadrado ()
DECLARE FUNCTION Raiz3 ()
COMMON SHARED n AS INTEGER 'Variable común y compartida en todos los módulos

CLS
INPUT "Introduce un entero -> ", n
PRINT : PRINT
COLOR 15
PRINT TAB(33); "1.- Cuadrado"
PRINT TAB(33); "2.- Raíz cubica aproximada"
PRINT : PRINT : COLOR 7
INPUT "Escoja su opción ->> ", op

SELECT CASE op
   CASE IS = 1
      resultado = Cuadrado
   CASE IS = 2
      resultado = Raiz3
   CASE ELSE
      COLOR 23
      PRINT "OPCION INVALIDA"
      END
END SELECT
PRINT "Resultado: "; resultado
```

```
'Modulo modtest.bas
```

```
COMMON SHARED n AS INTEGER      'También tenemos que declararla aquí
DECLARE FUNCTION Cuadrado ()
DECLARE FUNCTION Raiz3 ()

FUNCTION Cuadrado
  PRINT n
  Cuadrado = n * n
END FUNCTION

FUNCTION Raiz3
  DO WHILE (c * c * c < n)
    c = c + .0001
  LOOP
  Raiz3 = c
END FUNCTION
```

---

jafet_81@yahoo.com

Capítulo 6- Manipulación de cadenas de texto

La manipulación de cadenas es algo muy importante en programación, y QB nos proporciona muchas herramientas para modificar cadenas de una manera fácil y rápida. En QB podemos tener cosas como

```
destinatario$ = "Sr." + nombre$
```

cosa que no podemos hacer en lenguajes como C. Veamos el siguiente ejemplo:

```
' Ejemplo de suma de cadenas.

CONST saludo$ = "Bienvenido a QuickBASIC, "
CLS
INPUT "Introduce tu nombre: ", nombre$
PRINT : PRINT
PRINT saludo$ + nombre$
```

También podemos tener cosas como

```
IF cadena1$ < cadena2$ THEN...
```

En este caso la comparación se hace mediante el valor ASCII de cada cadena. Por citar un ejemplo si tenemos

```
'Comparación de cadenas
cadena1$ = "QuickBASIC"
cadena2$ = "GW-BASIC"

CLS
IF cadena1$ < cadena2$ THEN
   PRINT cadena1$; " es menor que "; cadena2$
ELSE
   PRINT cadena1$; " es mayor que "; cadena2$
END IF
END
```

veremos que los valores ASCII de los caracteres de cada cadena son:

Letra	Valor ASCII	Letra	Valor ASCII
Q	81	G	71
u	117	W	87
i	105	-	45
c	99	B	66
k	107	A	65
B	66	S	83
A	65	I	73
S	83	C	67
I	73		
C	67		
Suma	**863**	**Suma**	**557**

Nota: La tabla de los caracteres ASCII viene incluida en la ayuda de QuickBASIC 4.5 y MS-DOS QBASIC, consultala para mayor información.

Algo que veremos, es que podemos tener cadenas de un tamaño de caracteres definido por nosotros.

```
DIM nombre AS STRING * 6      'Definimos una variable que almacenara una
                              'cadena de 6 caracteres.
CLS
INPUT "Entra tu nombre completo: ", nombre
PRINT "Los primeros 6 caracteres de tu nombre completo son: " + nombre
END
```

Ahora veamos algunas **funciones** del lenguaje que nos permiten manejar cadenas de caracteres.

TOMANDO PARTES DE CADENAS

LEFT$(s$, n)

Función de cadena que devuelve los *n* caracteres de la cadena *s$* que están mas a la **izquierda**.

```
'Demuestra el uso de la función LEFT$
saludo$ = "HOLA A TODOS"

CLS
PRINT "La cadena completa es "; : COLOR 15: PRINT saludo$
result$ = LEFT$(saludo$, 5): COLOR 7
PRINT "Los 5 caracteres mas a la izq. de la cadena son "; : COLOR 15: PRINT result$
END
```

RIGHT$(s$, n)

Función de cadena que devuelve los *n* caracteres de la cadena *s$* que están mas a la **derecha**.

```
'Demuestra el uso de la función RIGHT$

saludo$ = "HOLA A TODOS"

CLS
PRINT "La cadena completa es "; : COLOR 15: PRINT saludo$
result$ = RIGHT$(saludo$, 5): COLOR 7
PRINT "Los 5 caracteres que están mas a la derecha de la cadena son "; : COLOR 15: PRINT
result$
END
```

MID$

Sintaxis 1 (como función):

Devuelve una subcadena a partir de una posición determinada de una cadena.

MID$(cadena$, inicio[,longitud])

- *cadena$* identifica a la cadena **de la cual** obtendremos una subcadena.
- *inicio* es una expresión numérica que especifica la posición desde donde obtendremos la subcadena.
- *longitud* puede ser omitida si quieres que la subcadena sean todos los caracteres hacia la derecha de la posición *inicio*.

```
'MID$ como función

cadena$ = "BASIC es fácil"
CLS
PRINT "La cadena completa: "; : COLOR 15: PRINT cadena$
COLOR 7
PRINT "Con MID$ desde la posición 10 :"; : COLOR 15: PRINT MID$(cadena$, 10)
END
```

Sintaxis 2 (como orden):

Sustituye una subcadena por otra.

MID$(c$, inicio[,longitud]) = cadena2$

- *c$* es una **variable** String.
- *inicio* es una expresión numérica que especifica la posición desde donde obtendremos la subcadena a substituir.
- *longitud* puede ser omitida si quieres que la subcadena sean todos los caracteres hacia la derecha de la posición *inicio*.
- *cadena2$* es la cadena que pondremos.

```
'MID$ como orden

cadena$ = "BASIC NO es fácil"    'cadena original
CLS
PRINT cadena$
COLOR 7: PRINT
MID$(cadena$, 7, 2) = "SI"          'modificando la cadena
PRINT cadena$                       'cadena resultante
```

LTRIM$(c$)

Función de cadena que le quita los espacios **iniciales** a c$ (si los tiene). Bien útil.

```
'Ejemplo de la utilisima función LTRIM$
cadena$ = "    cadena con espacios"    'cadena original

CLS
```

```
PRINT cadena$
COLOR 7: PRINT
csinesp$ = LTRIM$(cadena$)          'quitándole los espacios
PRINT csinesp$                      'cadena sin espacios
```

RTRIM$(c$)

Función de cadena que le quita los espacios **finales** a c$ (si los tiene). También útil.

```
'Ejemplo de la utilisima función RTRIM$

cadena$ = "cadena con espacios al final        "    'cadena original

CLS
PRINT cadena$; "(fin de la cadena)"
COLOR 7: PRINT
csinesp$ = RTRIM$(cadena$)                    'quitándole los espacios
PRINT csinesp$; "(fin de la cadena)"          'cadena sin espacios
END
```

BUSCANDO CADENAS

INSTR

Busca una cadena dentro de otra.

Sintaxis:

INSTR([inicio,] cadena_fuente, cadena_a_buscar)

- *inicio* es una expresión desde donde se empezaría a buscar la cadena_a_buscar.
- cadena_fuente es la cadena donde queremos buscar a cadena_a_buscar.
- cadena_a_buscar es la cadena que quermos encontrar.

```
'Usa la funcion INSTR

CLS
INPUT "Introduce una cadena que contenga el texto <<magi>>: ", c$
lugar = INSTR(c$, "magi")
IF lugar <> 0 THEN
   PRINT "<<magi>> encontrado en la posicion "; lugar
ELSE
   PRINT "La cadena no contiene a <<magi>>"
END IF
END
```

CONVIRTIENDO DE MAYUSCULAS A MINUSCULAS O VICEVERSA

UCASE$(cadena$)

Convierte a *cadena$* a **mayúsculas**. Muy útil.

```
'Usa la función UCASE$.
CLS
INPUT "Introduce una cadena: ", cadena$
PRINT "La cadena en mayúsculas es " + UCASE$(cadena$)
END
```

LCASE(cadena$)

Convierte a *cadena$* a **minúsculas**.

```
'Usa la función LCASE$.
CLS
INPUT "Introduce una cadena: ", cadena$
PRINT "La cadena en minúsculas es " + LCASE$(cadena$)
END
```

TAMAÑO DE LAS CADENAS

LEN(s$)

Es una muy util función numérica que devuelve la longitud de la cadena *s$*. Ejemplo

```
'Demuestra el uso de la funcion LEN.
'Valida si una palabra es palindromo o no. Un palindromo es una palabra
'que se lee igual al derecho que al reves, por ejemplo "solos", "oro", etc.

'$DYNAMIC            'Variables dinamicas
DIM letras$(1 TO 1)

DO
  'Cabeceras
  CLS
  COLOR 10
  PRINT TAB(26); "JUEGO DE LOS PALINDROMOMOS"
  PRINT : PRINT : COLOR 7

  'Fuerza al usuario a escribir algo B-)
  DO
    INPUT "Introduce una palabra: ", palabra$
    tam = LEN(palabra$)
  LOOP WHILE tam = 0

  'Nueva dimension del arreglo
  REDIM letras$(1 TO tam)
```

```
  'Guardamos las letras de la palabra en un arreglo
  FOR i = 1 TO tam
    letras$(i) = MID$(palabra$, i, 1)
  NEXT i

  'Damos nulo a alreves$ para que se reinicialize en cada bucle
  alreves$ = ""

  'Formamos a la palabra al reves
  FOR i = tam TO 1 STEP -1
    alreves$ = alreves$ + letras$(i)
  NEXT i

  'Verificamos si la palabra es igual al derecho que al reves
  IF palabra$ = alreves$ THEN
    PRINT "SI ES PALINDROMO"
  ELSE
    PRINT "NO ES PALINDROMO"
  END IF
  PRINT

  'Obligamos al usuario a introducir una opcion valida d:-)
  DO
    INPUT "¨OTRA PALBRA (s/n)"; otra$
  LOOP UNTIL UCASE$(otra$) = "S" XOR UCASE$(otra$) = "N"

LOOP UNTIL UCASE$(otra$) = "N"   'Si no quiere otra palabra terminamos.
END
```

CONVERSIONES ENTRE TEXTO Y NUMEROS

STR$(n)

Dvuelve a *n* como cadena, siendo n un valor numérico. Si *n* es positivo se agrega un espacio al principio; si tiene punto decimal seguido de **solo** ceros, se le quitan.

```
CLS
INPUT "Introduce un numero ->> ", n
PRINT "La raiz cuadrada de"; n; "es"; SQR(n)
numcadena$ = STR$(n)
IF INSTR(numcadena$, ".") THEN     'Si la función retorna un valor <> 0
  PRINT "El numero contiene decimales válidos"
ELSE
  PRINT "El numero no contiene decimales válidos"
END IF
END
```

VAL(n$)

Devuelve a *n$* como número. Este función terminará de inspeccionar a la cadena al momento en que halle valores no-numéricos ó esta se termine.

```
CLS
INPUT "Introduce tu fecha de naciemiento (dd/mm/aaaa): ", fecha$
anno = VAL(RIGHT$(fecha$, 4))

SELECT CASE anno
  CASE IS <= 1900
    PRINT "Naciste antes de, ó en, 1900"
  CASE IS <= 2000
    PRINT "Naciste antes de, ó en, 2000"
  CASE ELSE
    PRINT "Naciste despues de 2000"
END SELECT
END
```

ASC(c$)

Devuelve el **valor** ASCII del **primer** caracter de una cadena. Si la cadena es nula se produce un error en tiempo de ejecución *"Illegal function call"* ("Llamado de función ilegal").

```
CLS
INPUT "Introduce un caracter: ", car$
PRINT "El VALOR ASCII correspondiente es"; ASC(car$)
END
```

CHR$(n)

Devuelve el ASCII correspondiente a *n*. El valor de n debe estar entre 0 y 255.

```
mensaje$ = "Texto entre comillas dobles"
CLS
PRINT CHR$(34) + mensaje$ + CHR$(34)        'Caracter "
END
```

LEYENDO CARACTERES DEL TECLADO

INKEY$

Es una función que lee las **teclas** que presionamos. Las unicas teclas que son ignoradas son las que significan funciones especiales para el DOS o Windows, por ejemplo **Ctrl + C** (termina la ejecución de un programa), **Crtl + Alt + Sup** (reinicio del sistema), etc.

Un uso muy comun que se le da a esta funcion, es el esperar a que el usario presione una determinada tecla:

```
'Hacer mientras que la tecla pulsada sea ninguna ("")
DO : LOOP WHILE INKEY = ""          '("") es la cadena nula
```

'Hacer hasta que la tecla pulsada sea <ESC>
DO : LOOP UNTIL INKEY = CHR$(17) 'El caracter ASCII numero 17 es <ESC>

CADENAS DE CARACTERES IGUALES

SPACE$(n)

Devuelve una cadena de *n* espacios.

```
'Demuestra el uso de la funcion SPACE$
CLS
PRINT "PRESIONA <ENTER> PARA SALIR EN CUALQUIER MOMENTO..."
'Mientras tecla pulsada igual a ""
DO: LOOP WHILE INKEY$ = ""
DO
  'Pone los limites superiores e inferiores
  c = c + 1
  IF c MOD 2 = 0 THEN
    liminf = 0
    limsup = 5
    decre = -1
  ELSE
    liminf = 5
    limsup = 0
    decre = 1
  END IF
  'Selecciona el color a usar
  clavecolor = clavecolor + 1
  IF clavecolor MOD 2 = 0 THEN
    COLOR 15
  ELSEIF clavecolor MOD 3 = 0 THEN
    COLOR 4
    clavecolor = 0
  ELSE
    COLOR 2
  END IF
  'Pone el numero de espacios a partir de la columna 34 de la pantalla
  FOR i = limsup TO liminf STEP decre
    PRINT TAB(34); SPACE$(i) + CHR$(2) + CHR$(2) 'Carilla sonriente rellena.
  NEXT i

  FOR i = 1 TO 200000: NEXT  'Para que dure un tiempo. En TB puedes usar DELAY.

LOOP UNTIL INKEY$ = CHR$(13)  'INKEY$ lee una tecla pulsada. El caracter
                              'ASCII #13 corresponde a <ENTER>.
CLS
END
```

STRING$

Devuelve una cadena de un mismo caracter.

Sintaxis:

STRING$(n, c)

- n es el numero de caracteres que tendra la cadena
- c es el numero del codigo ASCII que corresponde al caracter que llevará la cadena. Debe ser un valor entero 0 - 255.

STRING$(n, c$)

- n es el numero de caracteres que tendra la cadena
- c$ es el caracter que llevará de la cadena. Si es una cadena de más de un caracter solo se tomará el primero.

```
CLS
PRINT TAB(20); STRING$(15, "*") + " CABECERA " + STRING$(15, 42)
END
```

---

jafet_81@yahoo.com

Capítulo 7: Manejo de archivos

Durante este capitulo veremos las maneras en que podemos manejar información mediante archivos (ficheros) de datos. El saber manejar archivos es muy importante ya que nos permite el almacenar información y además podremos consultarla o modificarla cada vez que queramos. QuickBASIC nos permite crear ficheros a los cuales podremos acceder de dos maneras: **secuencialmente** ó **aleatoriamente**. Pero bueno, antes que todo veamos como se estructuran los ficheros **de datos**:

Archivo (proveedores.dat)

	Campo 1	**Campo 2**	**Campo 3**	**Campo 4**
Registro 1	Aceros Serreño	Juan Almeida	(01-157)14-25-48	14785
Registro 2	FIECHICZA	Pedro Franco	(01-157)54-58-21	5000
Registro 3	COFIACZA	Patricio Celestino	13-20-20	20001
Registro 4	Química del Noroezte	Ramón Carballa	(01-157)85-20-17	12000
Registro 5	Servicios Metalúrgicoz	Omar Loya	15-50-95	21543
...	...	...	...	...

Como podemos ver, un **registro** es un conjunto de campos y por su parte un **campo** corresponde a un dato particular. La tabla anterior podría ilustrarnos un archivo de proveedores (*proveedores.dat)*, en el cual tenemos una serie de registros con sus respectivos campos, correspondiendo el campo 1 el nombre de la compañia, el campo 2 al nombre del contacto, el campo 3 al teléfono, y el campo 4 al monto que le debemos a cada uno.

ACCESO SECUENCIAL

Antes de nada, vamos a ver un simple ejemplo que muestra el contenido del autoexec.bat que se encuentra en nuestra computadora.

```
'AUTOEXEC.BAS
'Muestra el contenido del archivo AUTOEXEC.BAT

CLS
OPEN "c:\autoexec.bat" FOR INPUT AS #1      'Aquí puedes modificar la ruta, dependiendo de
                                            'donde tengas el autoexec.
DO WHILE NOT EOF(1)  'Hacer mientras no sea el fin del archivo
  LINE INPUT #1, linea$          'Guardamos la línea en una variable de cadena.
  PRINT linea$            'La mostramos en el monitor
LOOP                      'Vuelta
CLOSE #1                  'Debemos cerrar el archivo.
PRINT : COLOR 15
PRINT "FIN DEL ARCHIVO. PRESIONE CUALQUIER TECLA PARA CONTINUAR"
'Esperamos hasta oprimir una tecla
DO
LOOP UNTIL INKEY$ <> ""
END
```

El programa anterior trabaja de la siguiente manera:

- Abrimos el archivo para **leer de el** (FOR INPUT) y le ***damos*** el #1.

- Mientras no sea el fin del archivo #1 (**EOF(1)**, End Of File) leemos una linea del archivo y guardamos su contenido en la variable *linea$* (INPUT #1, linea$), para luego imprimirla en la pantalla.
- Cerramos el archivo.

En un archivo secuencial, los datos se procesan una **secuencia** determinada; del ejemplo anterior, tenemos que primero leemos la primera linea, luego la segunda, luego la tercera, etc. Ahora pasemos a ver las nuevas sentencias que utilizamos:

OPEN

Sintaxis básica:

OPEN *archivo* FOR *modo* AS *#numero_de_archivo*

Donde:

- La palabra *archivo* es una **cadena** que especifica la ubicación del archivo que queremos usar.
- La palabra *modo* indica la forma en que queremos utilizar el archivo:
  - **RANDOM:** El archivo se abrirá como **aleatorio**, es decir, para leer o escribir en una **posición especifica**.
  - **INPUT:** El fichero se abrirá para **leer** en el.
  - **OUTPUT**: El archivo se abré para **escribir** en él.
  - **APPEND**: El archivo será abierto para escribir al **final** de este.
- *El numero_de_archivo* especifica el **numero** que identificará al archivo dentro del programa.

Nota: Si el modo del archivo se omite, por default se accede como RANDOM.

LINE INPUT #

Lee una linea de texto desde un fichero.

Sintaxis:

INPUT *#numero_de_archivo, variable*

Donde:

- Con *numero_de_archivo* indicamos el numero de archivo del que queremos leer.
- Con *variable* indicamos la variable en donde guardaremos la linea del archivo.

EOF(n)

Esta función indica cuando es el fin del archivo.

CLOSE #n

Cierra el archivo *n* previamente abierto.

En el ejemplo anterior hemos abierto un fichero *sin formato*, es decir, un fichero que solo contiene lineas de texto. Ahora veamos como manejar un archivo que contenga datos...

```
'El siguiente programa crea un archivo y guarda datos en el para despues mos-
```

```
'trarlos.
'secalumn.bas

CLS
INPUT "Nombre del archivo (inclusive ruta y extension): ", FileName$
OPEN FileName$ FOR OUTPUT AS #1  'Abrimos el archivo para escribir
DO
   PRINT
   INPUT "Nombre del alumno: ", Nombre$
   INPUT "Matricula: ", Matricula$
   INPUT "Semestre: ", Semestre$
   INPUT "Grupo: ", Grupo$
   INPUT "Carrera: ", Carrera$
   'A continuacion escribimos CADENAS en el archivo
   WRITE #1, Nombre$, Matricula$, Semestre$, Grupo$, Carrera$
   PRINT : INPUT "¨Desea realizar otro registro (S/N)"; otro$
LOOP UNTIL UCASE$(otro$) = "N"
CLOSE #1: PRINT

OPEN FileName$ FOR INPUT AS #1  'Abrimos el archivo para leer
DO WHILE NOT EOF(1)
   CLS
   PRINT
   'Leemos los datos desde el archivo. Las variables tomarán el valor correc-
   'to ya que cada dato en el archivo esta entre comillas y separado por
   'comas.
   INPUT #1, Nombre$, Matricula$, Semestre$, Grupo$, Carrera$
   PRINT "Nombre del alumno: "; Nombre$
   PRINT "Matricula: "; Matricula$
   PRINT "Semestre: "; Semestre$
   PRINT "Grupo: "; Grupo$
   PRINT "Carrera: "; Carrera$
   DO: LOOP UNTIL INKEY$ <> ""
LOOP
CLOSE #1 'Cerramos el archivo
END
```

En el ejemplo anterior nos muestra algunas cosillas:

- Podemos dejar que el usuario introduzca el nombre y la ubicación del archivo, esto en tiempo de ejecución.
- La sentencia **WRITE #** escribe los datos en el archivo separados por comas para identificar los campos, después de cada coma es un nuevo campo.
- Para escribir en el archivo lo abrimos FOR OUTPUT y lo cerramos cuando terminamos de escribir. Luego, para leer de el lo volvemos a abrir pero ahora FOR INPUT y cuando terminamos lo cerramos.

Nota: La sentencia WRITE escribirá cada campo encerrado entre comillas dobles, lo cual indica que son cadenas de caracteres en forma de campos, separados por comas.

Aunque es algo muy facil y que no merece explicación, vamos a ver como copiar archivos ...

```
'Este programa muestra como copiar un archivo.
'Guardaremos datos sobre alumnos.
'copyfile.bas

DECLARE SUB GetData ()
DECLARE SUB ShowData ()
DECLARE SUB CopyData ()

'Variable compartida entre todos los procedimientos
COMMON SHARED OrigFileName$

‘---------------------- PROGRAMA PRINCIPAL ----------------------------------
GetData
ShowData
CopyData
‘-----------------------------------------------------------------------------------

SUB GetData
  CLS
  PRINT "GUARDANDO DATOS DE ALUMNOS": PRINT
  INPUT "Archivo inicial (ruta y extension si las hay): ", OrigFileName$
  OPEN OrigFileName$ FOR OUTPUT AS #1  'Abrimos el archivo para escribirlo
  DO
    PRINT
    INPUT "Nombre del alumno: ", Nombre$
    INPUT "Matricula: ", Matricula$
    INPUT "Semestre: ", Semestre$
    INPUT "Grupo: ", Grupo$
    INPUT "Carrera: ", Carrera$
    WRITE #1, Nombre$, Matricula$, Semestre$, Grupo$, Carrera$
    PRINT : INPUT "¨Desea realizar otro registro (S/N)"; otro$
  LOOP UNTIL UCASE$(otro$) = "N"
  CLOSE #1: PRINT
END SUB

SUB ShowData
  OPEN OrigFileName$ FOR INPUT AS #1  'Abrimos el archivo para leer.
  CLS
  PRINT "MOSTRANDO DATOS DE ALUMNOS CONTENIDOS EN "; OrigFileName$
  PRINT
  DO WHILE NOT EOF(1)
    c = c + 1
    COLOR 15: PRINT "REGISTRO #"; c: COLOR 7
    INPUT #1, Nombre$, Matricula$, Semestre$, Grupo$, Carrera$
    PRINT "Nombre del alumno: "; Nombre$
    PRINT "Matricula: "; Matricula$
    PRINT "Semestre: "; Semestre$
    PRINT "Grupo: "; Grupo$
    PRINT "Carrera: "; Carrera$
    PRINT
    PRINT "PRESIONA CUALQUIER TECLA..."
    DO: LOOP UNTIL INKEY$ <> ""
  LOOP
  CLOSE #1 'Cerramos el archivo
END SUB
```

```
SUB CopyData
  CLS
  INPUT "Nombre del archivo destino (incluyendo ruta y extension si la hay): ", DestFileName$
  OPEN OrigFileName$ FOR INPUT AS #1   'Abrimos el archivo origen para leer.
  OPEN DestFileName$ FOR OUTPUT AS #2  'Abrimos el archivo destino para escribirlo
  PRINT "COPIANDO "; OrigFileName$; " --> "; DestFileName$
  DO WHILE NOT EOF(1)
    INPUT #1, Nombre$, Matricula$, Semestre$, Grupo$, Carrera$
    WRITE #2, Nombre$, Matricula$, Semestre$, Grupo$, Carrera$
  LOOP
  PRINT : PRINT "HECHO"
  CLOSE #1: CLOSE #2
END SUB
```

ACCESO ALEATORIO

Como vimos anteriormente, en los archivos secuenciales tenemos que pasar por todos los registros para llegar a uno dado; por ejemplo, si queremos llegar al numero 5 necesitamos primero pasar por los registros del 1 al 4. En los **archivos aleatorios**, solamente necesitamos especificar la localización del registro para tener acceso a el.

TIPOS DE DATOS DEFINIDOS POR EL USUARIO.

Los tipos de datos definidos por el usuario nos permiten almacenar diferentes tipos de datos en las variables de este tipo, a diferencia de las variables sencillas que solo almacenan un tipo de dato. Un **tipo de dato definido por el usuario** se declara de la siguiente manera:

Sintaxis:

TYPE *nombre_del_tipo*
  campo AS tipo
  [campo AS tipo]
  ....
END TYPE

El *nombre_del_tipo* se usará para identificar variables con el nuevo formato. Los campos pueden ser de los tipos **predefinidos** (STRING, INTEGER, DOUBLE,...) o de otros **tipos definidos por el usuario**, pero declarados **antes**. Veamos el siguiente ejemplo:

```
'Ejemplo de datos definidos por el usuario.
'usrtype.bas

TYPE registro
  nombre AS STRING * 30     'El nombre debe caber en 40 caracteres
  matricula AS LONG         'La matricula como entero largo
  semestre AS INTEGER       'El grupo sera un entero
  grupo AS STRING * 1       'El grupo se identifica con un caracter
  carrera AS STRING * 3     'La carrera con 3 letras (LAE, ISC, IMM, etc)
END TYPE

DIM alumno AS registro      'Declaramos la variable alumno del TIPO registro

DO
```

```
   CLS
   INPUT "Nombre -> ", alumno.nombre
   INPUT "Matricula ->", alumno.matricula
   INPUT "Semestre -> ", alumno.semestre
   INPUT "Grupo -> ", alumno.grupo
   INPUT "Carrera -> ", alumno.carrera
   PRINT : INPUT "¨TODO BIEN? ", bien$
LOOP UNTIL UCASE$(bien$) = "S"
PRINT
PRINT "Nombre -> "; alumno.nombre
PRINT "Matricula ->"; alumno.matricula
PRINT "Semestre -> "; alumno.semestre
PRINT "Grupo -> "; alumno.grupo
PRINT "Carrera -> "; alumno.carrera
END
```

En la practica datos como la matricula no es necesario que declaren como numero ya que no realizaremos operaciones con ella, solo la incluí a manera de ejemplo. Como se ve del ejemplo anterior, declaramos una variable llamada *alumno,* del tipo de usuario *registro*. Para acceder a los campos de la variable debemos de poner el nombre de la variable seguido de un punto y el nombre del campo al que queremos acceder. Ahora, vemos que los campos son de tipos diversos como INTEGER, STRING, LONG, etc. támbien los campos pueden ser de tipos definidos por nosotros; si es así, el tipo que va dentro del campo debe ser declarado antes....

```
'Ejemplo de datos definidos por el usuario.
'usrtype.bas
TYPE examenes                    ‘Debemos de declarar el tipo antes de usarlo en el
 primerparcial AS SINGLE         ‘campo.
 segundoparcial AS SINGLE
 tercerparcial AS SINGLE
END TYPE

TYPE registro
   nombre AS STRING * 30     'El nombre debe caber en 40 caracteres
   matricula AS LONG         'La matricula como entero largo
   semestre AS INTEGER       'El grupo sera un entero
   grupo AS STRING * 1       'El grupo se identifica con un caracter
   calificaciones AS examenes ‘Calificaciones del tipo examenes ;-)
   carrera AS STRING * 3     'La carrera con 3 letras (LAE, ISC, IMM, etc)
END TYPE

DIM alumno AS registro       'Declaramos la variable alumno del TIPO registro

DO
   CLS
   INPUT "Nombre -> ", alumno.nombre
   INPUT "Matricula ->", alumno.matricula
   INPUT "Semestre -> ", alumno.semestre
   INPUT "Grupo -> ", alumno.grupo
   COLOR 15: PRINT "Calificaciones:": COLOR 7
   INPUT "Primer parcial: ", alumno.calificaciones.primerparcial
   INPUT "Segundo parcial: ", alumno.calificaciones.segundoparcial
   INPUT "Tercer parcial: ", alumno.calificaciones.tercerparcial
```

```
   INPUT "Carrera -> ", alumno.carrera
   PRINT : INPUT "¨TODO BIEN? ", bien$
LOOP UNTIL UCASE$(bien$) = "S"
PRINT
PRINT "Nombre -> "; alumno.nombre
PRINT "Matricula ->"; alumno.matricula
PRINT "Semestre -> "; alumno.semestre
PRINT "Grupo -> "; alumno.grupo
COLOR 15: PRINT "Calificaciones:": COLOR 7
PRINT "Primer parcial: "; alumno.calificaciones.primerparcial
PRINT "Segundo parcial: "; alumno.calificaciones.segundoparcial
PRINT "Tercer parcial: "; alumno.calificaciones.tercerparcial
PRINT "Carrera -> "; alumno.carrera
END
```

Como vemos, el operador punto se asocia de izquierda a derecha, por ejemplo en *alumno.calificaciones.primerparcial* el compilador primero asocia el campo *primerparcial* con *calificaciones*, y luego *calificaciones* con alumno. Confío en que el ejemplo se explique por si solo. Como cualquier tipo de dato, podemos tambien tener **arreglos** de tipos de datos definidos por el usuario, por ejemplo, si tenemos algo como:

```
TYPE registro
   nombre AS STRING * 30     'El nombre debe caber en 40 caracteres
   matricula AS LONG         'La matricula como entero largo
   semestre AS INTEGER       'El grupo sera un entero
   grupo AS STRING * 1       'El grupo se identifica con un caracter
   carrera AS STRING * 3     'La carrera con 3 letras (LAE, ISC, IMM, etc)
END TYPE

DIM GrupoA(1 TO 30) AS registro  'Declaramos un arreglo de tipo registro
....
```

y queremos cambiar la matricula del alumno que se encuentra en la posición 4 del arreglo, accederemos este dato poniendo

```
GrupoA(4).matricula = 150424  'Le damos el valor de 150424
```

ARCHIVOS DE ACCESO ALEATORIO

Para abrir un archivo para acceso aleatorio usamos:

Sintaxis basica:

OPEN nombre_del_archivo FOR RANDOM [ACCESS acceso] [LEN = longregistro]

Donde

- **RANDOM** especifica que es de acceso aleatorio, y la clausula opcional **ACCESS** nos permite especificar si queremos abrirlo para lectura (**READ**), escritura (**WRITE**) o **ambos (READ WRITE**).
- Si se omite la clausula ACCESS, se producen 3 intentos para su acceso:
  1) Se intenta acceder para escribir y leer (READ WRITE)
  2) Se intenta acceder para escribir solamente (WRITE)
  3) Se intenta acceder para leer solamente (READ)

Si los 3 intentos fallan se produce un error en tiempo de ejecución.

- La clausula **LEN** = l*ongregistro* nos permite especificar el tamaño de los registros y tambien el formato que tendrá el fichero, *longregistro* especifica el **numero de bytes** que le concederemos a cada registro.

Veamos el siguiente ejemplo:

```
'Manejo de archivos de acceso aleatorio.
'filerdn.bas

TYPE registro
   nombre AS STRING * 30     'El nombre debe caber en 40 caracteres
   matricula AS LONG         'La matricula como entero largo
   semestre AS INTEGER       'El grupo sera un entero
   grupo AS STRING * 1       'El grupo se identifica con un caracter
   carrera AS STRING * 3     'La carrera con 3 letras (LAE, ISC, IMM, etc)
END TYPE

DIM alumno AS registro, muestra AS registro

CLS
COLOR 15: PRINT TAB(30); "REGISTRO DE ALUMNOS": COLOR 7: PRINT
INPUT "¨Cuantos alumnos deseas registrar? ", n
'Abrimos el archivo aleatorio para leer y escribir en el.
'Supongo que la unidad C: exite, si no cambia la ruta.
OPEN "c:\alumnos.dat" FOR RANDOM AS #1 LEN = LEN(alumno)
FOR i = 1 TO n
   DO
      INPUT "Nombre -> ", alumno.nombre
      INPUT "Matricula -> ", alumno.matricula
      INPUT "Semestre -> ", alumno.semestre
      INPUT "Grupo -> ", alumno.grupo
      INPUT "Carrera -> ", alumno.carrera
      PRINT : INPUT "¨TODO BIEN? ", bien$
   LOOP UNTIL UCASE$(bien$) = "S"
   'Escribimos el contenido de la variable alumno en la posicion i del
   'archivo.
   PUT #1, i, alumno
   COLOR 15
   PRINT "ALUMNO ALMACENADO EN EL REGISTRO"; i; ". PRESIONA CUALQUIER TECLA"
   DO: LOOP WHILE INKEY$ = ""
NEXT i
PRINT

CLS
INPUT "¨Que numero de registro quieres ver? ", regnumero
PRINT : COLOR 7
GET #1, regnumero, muestra    'Tomamos el registro en la posicion dada por
                              'el usuario y guardamos su contenido en la
                              'variable muestra.
PRINT "Nombre -> "; muestra.nombre
PRINT "Matricula -> "; muestra.matricula
PRINT "Semestre -> "; muestra.semestre
```

```
PRINT "Grupo -> "; muestra.grupo
PRINT "Carrera -> "; muestra.carrera
CLOSE #1                'Cerramos el archivo.
KILL "c:\alumnos.dat"     'Borramos el archivo del disco duro
END
```

Del ejemplo anterior podemos comentar muchas cosas:

- Abrimos el archivo de acceso aleatorio y le damos un "formato". A cada **registro** le damos la longitud en bytes de la variable *alumno* y cada **campo** del archivo coincidirá con los campos de la variable. La función LEN nos devuelve el **tamaño en bytes** que ocupa una **variable**.
- La sentencia **PUT #** nos permite "vaciar" el contenido de una variable en un registro del archivo.

Sintaxis:

PUT #numero_de_archivo[[, numero_de_registro][, variable]]

- La sentencia **GET #** nos permite "tomar" el contenido de un registro del archivo y guardarlo en una variable.

Sintaxis:

GET #numero_de_archivo[[, numero_de_registro][, variable]]

- El objetivo principal del ejemplo es mostrar que para tomar un registro no necesitamos procesar todos los registros, simplemente indicamos que numero de registro deseamos y GET lo tomará y lo guardará.
- La sentencia KILL borra un archivo del disco duro. Ojo...

OTRAS SENTENCIAS UTILES

Ya visto como se manejan basicamente los archivos, veamos algunas funciones y sentencias que son muy utilies en la manipulación de estos.

LOF(n)

Esta función retorna la longitud en bytes de el archivo numero *n*. Esta función es especialmente util, por ejemplo cuando queremos saber el numero de registros que tiene un archivo de acceso aleatorio con formato.

LOC(n)

Proporciona la posición del puntero en un archivo, es decir la linea o el registro en que se encuentra en ese momento.

EOF(n)

Esta función indica cuando es el fin del archivo *n*.

KILL

Esta sentencia borra un archivo del disco duro (o dispositivo).

Sintaxis:
KILL *nombrearchivo*

nombrearchivo puede incluir ruta y extensión.

RMDIR
Esta orden remueve un directorio del disco duro (o dispositivo), es equivalente al comado RD de DOS. El directorio debe de estar vacio o se produce un mensaje de error.

Sintaxis:
RMDDIR *nombredirectorio*

nombrdirectorio puede incluir ruta, p.e RMDIR "c:\windows\system"

MKDDIR
Esta orden crea un nuevo directorio en disco, es equivalente al comado MD de DOS.
Sintaxis:
RMDDIR *nombredirectorio*

nombredirectorio puede incluir ruta, p.e RMDIR "c:\qb45\qbfiles\modulos"

CHDIR
Esta orden cambia del directorio actual a un nuevo directorio, es equivalente al comado CD de DOS.
Sintaxis:
CHDIR *nombredirectorio*

nombredirectorio puede incluir ruta, p.e CHDIR "c:\" 'Nos vamos al directorio raiz de C:

NAME
Cambia el nombre de un directorio o de un archivo grabado en un disco.
Sintaxis:
NAME *viejonombre* AS *nuevonombre*

viejonombre y nuevonombre pueden contener una ruta.

LOCATE
Posiciona el cursor para escribir en una cordenada de la pantalla.

Sintaxis basica:
LOCATE fila, columna

LPRINT
Lo mismo que PRINT, salvo que la información será mandada a una **impresora** en vez de al monitor.

Nota: El equivalente en caracter al ASCII #12 es el **salto de pagina**, por lo que LPRINT CHR$(12) obligará a la impresorá a saltar a la siguiente hoja. El ASCII #13 es **<ENTER>** y el #27 **<ESC>**

Capítulo 8: Gráficos y sonidos

En este capítulo vamos a ver las distintas posibilidades gráficas que QuickBASIC nos permite, además de algunas instrucciones para emitir sonidos. La utilización de gráficos y sonidos en nuestros programas nos permite crear un ambiente mejor para quien vaya a tener que usar nuestros programas. Antes de trabajar con gráficos, vamos a ver algunas cosas útiles:

Gráficos vs. Texto

Hasta ahora solo hemos trabajado la pantalla en **modo texto**, es decir, escribiendo caracteres en ella. Ahora, por ejemplo, imaginemos que una maestra de estadística quiere entretenernos en lo que ella toma su café y nos pide que le mostremos en la computadora los resultados acerca una encuesta relativa a las edades de los alumnos del III semestre de la carrera de Ingeniería Geológica. Lo más sencillo sería hacer un programa que procese los datos y muestre lo siguiente:

muestra = población = 35 alumnos

Edad	Alumnos
18 – 20	15
21 – 23	19
24 – 26	1

media = 21.22, moda = 22, mediana = 22.5

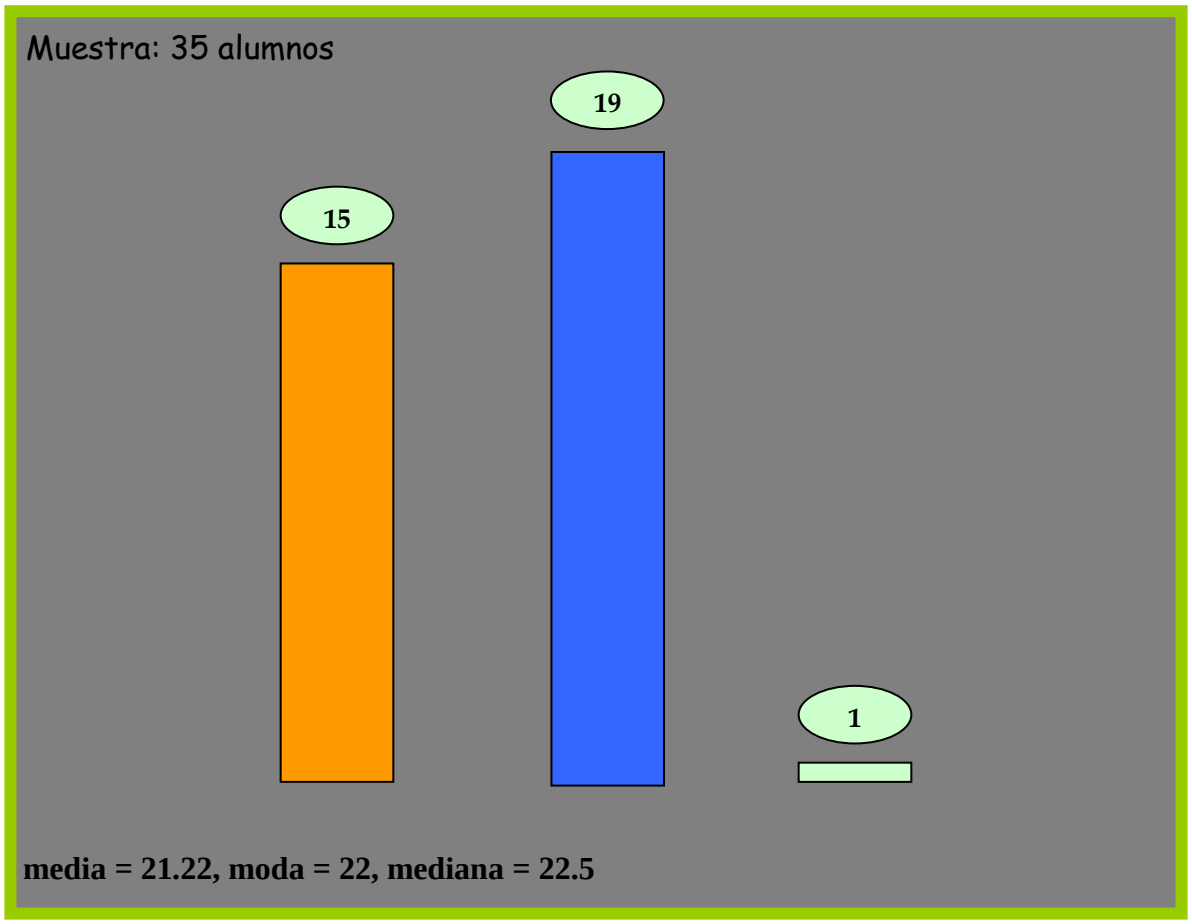
Al ver lo que hemos hecho, la maestra nos pide ahora que le mostremos los resultados en un gráfico de barras. O.K, entonces le podríamos mostrar algo como esto:

Muestra: 35 alumnos

```
                19
                ####
                ####
                ####
     15         ####
    ####        ####
    ####        ####
    ####        ####
    ####        ####
    ####        ####
    ####        ####
    ####        ####
    ####        ####        1
    ####        ####      ####
    18-20       21-23     24-26
```

media = 21.22, moda = 22 mediana = 22.5

Esto esta bien, se vale usar caracteres para crear las barras. Pero... ¿no sería mejor si le mostramos algo como lo siguiente?



Tal vez quedaría mas contenta.

Cuando trabajamos en **modo texto**, solo podemos presentar **caracteres y símbolos** en la pantalla; pero cuando trabajamos en **modo gráfico**, podemos acceder a cada uno de los **puntos** de la pantalla, llamados **pixeles** (*picture elements*, elementos de imagen). La combinación de varios pixeles nos permite crear formas y dibujos complejos.

Nota: Cabe recordar que, ya que los caracteres mismos estan formados por pixeles, dentro de un modo gráfico podemos trabajar con texto y además manejar formas gráficas, todo **al mismo tiempo**.

Otro concepto importante, además de pixel, es el de **resolución**. La resolución de la pantalla para mostrar gráficos esta dada por el número de pixeles que pueden mostrarse horizontalmente y verticalmente al mismo tiempo. Por ejemplo, podemos tener resoluciones de 320 x 200 pixeles, 640 x 200, 640 x 480, etc.; lógicamente que entre mayor sea la resolución, más pequeños serán los pixeles y por tanto la imagen será mas fina..

Algunos de los estándares de tarjetas gráficas son:

La IBM Monochrome Display and Printer Adapter (MDPA):
Permite el uso de un monitor monocromo para texto de 80 (columnas) x 25 (renglones)

La IBM Color Graphics Adapter (CGA):
Texto: 80x25 y 40x25 , con 16 colores
Gráficos: 320x200 en 4 colores, 640x200 en b/n.

La IBM Enhanced Graphics Adapter (EGA):
Compatible con CGA, pero incluye los modos gráficos:
- 320x200, 640x200 y 640x350 en 16 colores.

La IBM Multicolor Graphics Array (MCGA):
Además de emular la EGA, nos da también:
- 320x200 en 256 colores y 640x480 monocromo.

La IBM Video Graphics Array (VGA):
Además de incluir la MCGA, nos da:
- 640x480 en 16 colores.

La IBM Super Video Graphics Array (SVGA):
Igual que VGA, y puede incluir uno o más de los siguientes modos (dependiendo de quien las haga):
- 256 colores en 640x480, 800x600, 1024x768, 1280x1024... y hasta 32000, 64000 o 16 millones de colores.
- La llamada VESA, es un estándar para que todas sean lo mas parecido posible.

Hercules Graphics Card:
Texto: Igual que MDPA en texto.
Gráficos: 720x350 blanco y negro (b/n).

SCREEN

Podemos seleccionar el **modo** (combinación de resolución y colores) en que queremos trabajar mediante la sentencia **SCREEN**.

Sintaxis básica:
SCREEN *modo*

- Donde *modo* es un entero de 0 – 13 que especifica el modo en que queremos utilizar la pantalla para gráficos.

Nota: SCREEN 0 se refiere al modo de solo texto.

Para una información más detallada de la sentencia SCREEN consulta la ayuda de QB. Por cuestiones de compatibilidad solo me centrare en el uso de SCREEN 1 y 2, que es el que se usa para una tarjeta CGA, la mas básica para usar gráficos. Lógicamente que una tarjeta mas potente es compatible con CGA.

Si se selecciona SCREEN 1, tendremos un modo de pantalla de 320x200 pixeles con 16 colores de fondo y dos paletas con 3 colores c/u. Si se selecciona SCREEN 2, tendremos una resolución de 640x200 pixeles en fondo negro y formas blancas.

O.K., empecemos a trabajar con SCREEN 1. Como ya se mencionó, podemos tener una resolución de 320x200 y los colores de **fondo** y la **paleta** se especifican mediante la sentencia COLOR. En modo gráfico, la funcion COLOR funciona diferente a como lo hace en modo texto. En este caso COLOR tendrá 2 parametros: COLOR *fondo*, *paleta;* donde *fondo* es un entero de 0 – 15 que utilizaremos para especificar el color del fondo de la pantalla. Luego, *paleta* es un numero 0-1 que especifica 3 colores (además del color de de fondo) de que podemos disponer **al momento de dibujar nuestras formas**. Los colores dentro de cada **paleta** son los siguientes:

Paleta 0		Paleta 1	
0	Color de fondo	0	Color de fondo
1	Verde	1	Cyan
2	Rojo	2	Magenta
3	Marrón	3	Blanco

El color de **fondo** corresponde a:

No.	Color
0	Negro
1	Azul
2	Verde
3	Cyan
4	Rojo
5	Magenta
6	Marrón
7	Blanco
8	Gris
9	Azul claro
10	Verde claro
11	Cyan claro
12	Rojo claro
13	Magenta claro
14	Amarillo
15	Blanco brillante

Para que quede mas claro, veamos el siguiente ejemplo:

```
SCREEN 1        'Resolución de 320x200 pixeles
COLOR 2, 0      'Fondo azul (2) y paleta 0.
LINE (20, 20)-(200, 100), 3      'Línea que va desde las coordenadas (20,20) a (200,100) de la
                                 'pantalla y del color 3 de la paleta elegida (en este caso el color es
                                 marrón).
END
```

LINE

En el programa anterior podemos observar el uso de la sentencia **LINE**, la cual nos permite dibujar líneas y rectángulos.

Sintaxis básica:

LINE (x1, y1) – (x2, y2)[, color][, B[F]]

- Las coordenadas (x1, y1) especifican la posición desde donde saldrá nuestra línea, mientras que (x2,y2) las coordenadas donde terminará.
- El parametro opcional *color* es el numero que especifica el color de la línea. El valor de este parámetro se interpretará como uno de los colores de la paleta activa. Si se omite, se seleccionará de forma automática el color numero 3 de la paleta.
- El parámetro opcional B ("Box") hace que en vez de una línea, se dibuje un **contorno** rectangular cuya esquina superior izquierda estará en (x1, y1) y la esquina inferior derecha en el punto (x2, y2).
- El parámetro opcional F ("Fill") después de B, hace que se dibuje un rectángulo **sólido**. El color del rectángulo esta definido por el parámetro *color*.

Veamos el siguiente ejemplo:

```
SCREEN 1: COLOR 12, 1
LINE (10, 10)-(100, 100), , B       'Dibuja un contorno rectangular.
LINE (10, 140)-(300, 160), 2, BF    'Dibuja un rectángulo sólido.
END
```

PSET Y PRESET

Ahora veamos otra sentencia importante: **PSET**. La sentencia PSET no hace otra cosa sino dibujar un pixel en las coordenadas que le indiquemos y del color que le indiquemos. Los colores están en función de la paleta actual.

Sintaxis:

PSET (x, y) [, color]

- Las coordenadas *x, y* especifican la posición donde se dibujara el pixel.
- El parámetro opcional *color* es se interpretará como uno de los colores de la paleta activa. Si se omite, entonces se seleccionara de forma automática el color numero 3 de la paleta.

Otra sentencia importante es **PRESET**. Esta sentencia funciona de manera idéntica a PSET, salvo que si el color se omite, se seleccionará de forma automática el color de fondo. En

algunos programas puede resultar útil usar PSET para generar puntos y PRESET para borrarlos. Veamos el siguiente ejemplo:

```
SCREEN 1: COLOR 4, 1    'Color de fondo rojo y paleta 1
FOR i = 100 TO 150
  PSET (i, i)         'Dibujamos una línea con PSET
NEXT i

PRINT "PRESIONA UNA TECLA PARA BORRAR LA MITAD DE LA LINEA."
WHILE INKEY$ = "": WEND
PRINT

FOR i = 100 TO 150
  PRESET (i, i)       'Usamos PRESET para borrar la línea que hizo PSET
  IF i = 125 THEN
    PRINT "PRESIONA UNA TECLA PARA TERMINAR"
    WHILE INKEY$ = "": WEND
  END IF
NEXT i
END
```

CIRCLE

La sentencia **CIRCLE** nos permite dibujar un circulo en la pantalla.

Sintaxis básica:

CIRCLE (x, y), radio [,color]

- Las coordenadas x, y indican el **origen** del circulo.
- El parámetro *radio* es el valor que tendrá el radio del circulo (en pixeles)
- El parámetro opcional *color* indica el color del perimetro de la circunferencia.

Veamos el siguiente ejemplo:

```
SCREEN 1 'Como no especificamos ni fondo ni paleta, automáticamente el fondo
         'es 0 y la paleta 1
CIRCLE (160, 100), 100, 2 'Dibujamos un círculo con un radio de 100 pixeles
END
```

Otro ejemplo:

```
SCREEN 1

CONST maxx = 320, maxy = 200  'Coordenadas máximas como constantes

FOR i = 0 TO maxy / 2
  CIRCLE (maxx / 2, maxy / 2), i
  FOR time = 1 TO 120000: NEXT time: CLS 'Para que dure un tiempo y luego
                                         'limpie la pantalla para dibujar
                                         'el siguiente circulo.
```

```
  IF INKEY$ <> "" THEN END
NEXT i
```

PAINT

La sentencia **PAINT** nos permite rellenar.

Sintaxis Básica:

PAINT (x,y), color, color_de _borde

- Las coordenadas (x,y) se refieren al punto desde donde se empezará a expandir el relleno. Ojo que este punto debe de estar DENTRO de la forma que queremos rellenar.
- El parámetro *color* es un numero que especifica el color con el que queremos rellenar la forma. Debe de estar en función de la paleta activa.
- El *color_de_borde* es el numero del color del borde que queremos que detenga el proceso de relleno. Debe de estar en función de la paleta activa.

Veamos el siguiente ejemplillo para que quede mas claro:

```
'Ejemplo del uso de PAINT

'Pantalla en 320x200
SCREEN 1

'Guardamos en constantes las coordenadas máximas de la pantalla.
CONST maxx = 320, maxy = 200

'Circulo con origen en el centro de la pantalla, radio = 100, y color blanco
CIRCLE (maxx / 2, maxy / 2), 100, 3

PAINT (maxx / 2, maxy / 2), 1, 3
END
```

Veamos ahora otro ejemplo:

```
'Ejemplo del uso de PAINT

'Pantalla en 320x200
SCREEN 1

'Guardamos en constantes las coordenadas máximas de la pantalla.
CONST maxx = 320, maxy = 200

'Circulo con origen en el centro de la pantalla, de radio = 100
CIRCLE (maxx / 2, maxy / 2), 100, 1

PAINT (270, 1), 2, 1
END
```

GET Y PUT

Ahora analicemos dos de las sentencias mas importantes en el manejo de gráficos y animaciones: **GET** y **PUT**. La primera, GET, nos permite almacenar una imagen dentro de un arreglo numérico. La segunda, PUT, nos permite vaciar una imagen almacenada en un arreglo numérico.

Sintaxis Básica:

GET (x1, y1) – (x2, y2), arreglo

- Como en realidad lo que se almacenará será un área rectangular de la pantalla, las coordenadas (x1, y1) y (x2, y2) corresponden a las esquinas superior izquierda e inferior derecha del rectángulo que contendrá la imagen que queremos guardar.
- El parámetro *arreglo* se refiere al **nombre** del arreglo numérico donde se guardará la imagen. Es preferible que sea entero, por eso de la rapidez de los cálculos y para evitar errores cuando se ejecute el programa (error en tiempo de ejecución, *run-time error*).

Una pregunta que casi creo que muchos se estan haciendo es: ¿De que tamaño tiene que ser el arreglo para que guarde la imagen correctamente? Y la respuesta es: el tamaño ***exacto*** en **bytes** que necesita el arreglo para almacenar la imagen completa esta dado por la siguiente fórmula:

4 + INT(((x2 - x1 + 1) * (bits_por_pixel_por_plano) + 7)/8) * planos * ((y2 - y1) + 1)

Algo complicada, ¿no? Los *bits_por_pixel_por_plano* y los *planos* estan dependen del modo en que usemos SCREEN. La siguiente tabla muestra los bits por pixel por plano y los planos para cada modo de pantalla:

Valores para bits por pixel por plano y para planos		
Modo de SCREEN	Bits por pixel	Planos
1	2	1
2	1	1
7	1	4
8	1	4
9	1	2 Si hay 64k de memoria EGA
	1	4 Si hay > 64k de memoria EGA
10	1	2
11	1	1
12	1	4
13	8	1

Como ya sabemos, un entero corto ocupa 2 bytes, un entero largo 4 bytes, un real simple 4 bytes, y uno de doble precisión 8 bytes. Así, por ejemplo, un arreglo de 3 enteros largos ocupará 12 bytes.

Por ejemplo, supongamos que queremos usar GET para guardar una imagen hecha usando SCREEN 2. Si las coordenadas de la esquina superior izquierda de la imagen son (0,0), y las

coordenadas de la esquina inferior derecha son (32,32), entonces el tamaño exacto, en **bytes**, del arreglo es

4 + INT((33 * 1 + 7)/8) * 1 * (33)

O 169. Esto quiere decir que un arreglo con 85 elementos INTEGER quedaría más que perfecto para guardar la imagen. Lo anterior es para quien desee obtener un tamaño exacto, yo personalmente prefiero aproximar el arreglo; cuando el arreglo no alcanza a almacenar la imagen se produce un error ***Illegal Function Call***. (*Llamada de función inválido*). Ahora sigamos con PUT:

Sintaxis Básica:

PUT (x, y), arreglo

- La coordenada (x, y) corresponde al punto donde irá la esquina superior izquierda del rectángulo que contendrá a la imagen que vamos a poner.
- El parámetro *arreglo* se refiere al **nombre** del arreglo numérico donde se guardó previamente la imagen.

Un ejemplo:

```
'Pantalla en modo 320x200
SCREEN 1
'Declaramos las coordenadas mas importantes (limite en x, limite en y,
'mitad en x de la pantalla, mitad en y)
CONST maxx = 320, maxy = 200, mitadx = maxx / 2, mitady = maxy / 2

'El arreglo en el que almacenaremos la figura, en este caso una "pelota"
DIM pelota(70) AS INTEGER

'Dibujamos la pelota
CIRCLE (mitadx, mitady), 10, 2
PAINT (mitadx, mitady), 2, 2
'La guardamos en el arreglo
GET (mitadx - 10, mitady - 10)-(mitadx + 10, mitady + 10), pelota

PRINT : PRINT
PRINT TAB(5); "PRESIONA CUALQUIER TECLA PARA COPIAR LA PELOTA A (0,0)"
WHILE INKEY$ = "": WEND

'Dibujamos la pelota en (0,0)
PUT (0, 0), pelota

PRINT
PRINT TAB(5); "PRESIONA CUALQUIER TECLA PARA BORRAR LA PELOTA CENTRAL"
WHILE INKEY$ = "": WEND

'Cuando ponemos una imagen sobre si misma, esta borra.
PUT (mitadx - 10, mitady - 10), pelota
END
```

Por último veamos otro ejemplo:

```
DECLARE SUB CopiarCarro ()
DECLARE SUB BorrarCarro ()
DECLARE SUB MoverCarro ()
DECLARE SUB DibujarCarro ()

SCREEN 1

CONST maxx = 320, maxy = 200, mitadx = maxx / 2, mitady = maxy / 2
DIM SHARED carro(70) AS INTEGER

DibujarCarro
CopiarCarro
BorrarCarro
MoverCarro

SUB DibujarCarro
  LINE (mitadx, mitady)-(mitadx + 30, mitady + 5), 2, BF
  LINE (mitadx, mitady - 5)-(mitadx + 20, mitady), 1, BF
  'Rueda atrás
  CIRCLE (mitadx + 5, mitady + 7), 4, 3
  PAINT (mitadx + 5, mitady + 7), 3, 3
  'Rueda adelante
  CIRCLE (mitadx + 25, mitady + 7), 4, 3
  PAINT (mitadx + 25, mitady + 7), 3, 3
END SUB

SUB CopiarCarro
  GET (mitadx, mitady - 5)-(mitadx + 30, mitady + 10), carro
END SUB

SUB BorrarCarro
  PUT (mitadx, mitady - 5), carro
END SUB

SUB MoverCarro
  x = 0: y = 0
  FOR i = 1 TO maxx - 30
    PUT (x, y), carro
    FOR c = 1 TO 15000: NEXT c ‘Para que se detenga un poco.
    PUT (x, y), carro
    x = x + 1
  NEXT i
END SUB
```

VIEW Y WINDOW

La sentencia VIEW define una nueva pantalla para la visualización de gráficos (**puerto visualizador de gráficos**).

Sintaxis Básica:

VIEW [SCREEN][(x1, y1) – (x2, y2)[,[color][,borde]]

- Si la sentencia VIEW aparece sin argumentos, toda la pantalla actuará como puerto visualizador, que es lo normal.
- Las coordenadas (x1, y1) y (x2, y2) corresponden a las esquinas superior izquierda e inferior derecha del área rectangular que definirá el nuevo puerto visualizador de gráficos. Fuera de esta área NO se podrán visualizar gráficos.
- Los parámetros opcionales *color* y *borde* se refieren al color de relleno del puerto visualizador y al color del borde respectivamente.
- El modificador SCREEN se usa para indicar que la sentencia VIEW actuará en función de toda la pantalla, y no en función de otras sentencias VIEW si las hubiera.

Veamos un ejemplo:

```
'320x200
SCREEN 1

'El nuevo puerto visualizador será desde (100, 50) a (250, 150), sin relleno
'y con borde blanco. Fuera de esta área no se podrán visualizar gráficos.
VIEW (100, 50)-(250, 150), , 3
'Las coordenadas estarán en función DEL PUERTO VISUALIZADOR.
CIRCLE (100, 50), 10, 1
END
```

Ahora veamos el ejemplo anterior, pero usando el modificador SCREEN:

```
'320x200
SCREEN 1

'El nuevo puerto visualizador será desde (100, 50) a (250, 150), sin relleno
'y con borde blanco. Fuera de esta área no se podrán visualizar gráficos.
VIEW SCREEN (100, 50)-(250, 150), , 3
'Como usamos el modificador SCREEN, las coordenadas estarán en función de TODA LA
'PANTALLA.
CIRCLE (100, 50), 10, 1
END
```

La sentencia **WINDOW**, crea un **nuevo sistema de coordenadas.**

Sintaxis Básica:

WINDOW [[SCREEN][(x1, y1) – (x2, y2)]

- Si la sentencia WINDOW aparece sin argumentos, el sistema de coordenadas será el mismo que usa la pantalla por defecto, de acuerdo a la sentencia SCREEN
- Los puntos (x1, y1) y (x2, y2) definirán el **límite inferior** y el **límite superior** del nuevo sistema de coordenadas. El sistema de coordenadas irá de negativo a positivo a partir de la esquina inferior izquierda de la pantalla.
- El parámetro opcional SCREEN invierte la dirección normal de los puntos del eje "y".

Para que quede más conciso, veamos el siguiente ejemplo:

```
SCREEN 1
```

```
'El límite inferior del nuevo sistema de coordenadas será en (0,0)
'y el límite superior en (200,150)
WINDOW (0, 0)-(200, 150)
'Recordemos que el nuevo sistema de coordenadas partirá desde la esquina inferior izquierda de la
'pantalla
LINE (0, 0)-(100, 75), 3
END
```

La sentencia WINDOW tiene la ventaja de que nos permite crear sistemas de coordenadas en el que podamos manejar localizaciones **negativas** y **decimales**, por ejemplo: (-2, 4.6), (-150,-20), (120.1,-15), etc. Veamos...

```
'El siguiente ejemplo muestra como la sentencia WINDOW nos permite crear sistemas
'de coordenadas con localizaciones negativas. Vamos a graficar la función
'la función y = x² desde x = -10 hasta x = 10

SCREEN 1

'Limite inferior del sistema = -10 en x, -100 en y
'Limite superior del sistema = 10 en x, 100 en y
WINDOW (-10, -100)-(10, 100)

'Dibujamos el eje x
LINE (-10, 0)-(10, 0), 3

'Dibujamos el eje y
LINE (0, -100)-(0, 100), 3

'Graficamos de x = -10 hasta 10, de .05 en .05
FOR x = -10 TO 10 STEP .05
   y = x * x
   PSET (x, y), 1
NEXT x
END
```

Por último veamos como combinar WINDOW y VIEW:

```
'320x200
SCREEN 1
PRINT TAB(18); "y = x²"

'Definimos el área donde se verán los gráficos ("pantalla").
VIEW (120, 20)-(200, 100)

'Creamos el nuevo sistema de coordenadas, el cual estar  en función de la
'nueva "pantalla" creada con VIEW
WINDOW (0, 0)-(10, 100)

'Dibujamos los ejes
LINE (0, 100)-(0, 0), 3: LINE (0, 0)-(100, 0), 3
```

```
'Marcadores del eje x
FOR x = 0 TO 10 STEP 2
   LINE (x, 2)-(x, 0), 3
   LOCATE 14, 15 + x: PRINT x
NEXT x

'Marcadores del eje y.
FOR y = 0 TO 100 STEP 20
   LINE (0, y)-(.5, y), 3  'Nótese el decimal
   IF y <> 0 THEN LOCATE 13 - y / 10, 11: PRINT y
NEXT y

'Gráfica
FOR x = 0 TO 10 STEP .05
   y = x * x ‘Si deseas ver otras gráficas solo cambia la función
   PSET (x, y), 1
NEXT x
END
```

VIEW PRINT

Esta sentencia define una nueva área para la visualización de texto (**puerto de visualización de texto)**.

Sintaxis:

VIEW PRINT [renglonsuperior, rengloninferior]

- El parámetro opcional *renglonsuperior,* indica el número del renglón **desde** donde se empezará a ver el texto.
- El parámetro opcional *rengloninferior,* indica el número del renglón **hasta** donde se verá el texto.

A ver el siguiente ejemplillo.....

```
VIEW PRINT 10 TO 20
DO
   i = i + 1
   PRINT "QuickBASIC es potente... "; TAB(60); i
   IF i = 32767 THEN i = 0
LOOP WHILE INKEY$ = ""
END
```

O. K, ahora watchemos otro ejemplillo en el que definimos un puerto para gráficos y otro para texto...

```
CONST maxx = 320, maxy = 200
SCREEN 1
RANDOMIZE TIMER
```

```
' Fijamos las coordenadas del puerto de visualización de GRAFICOS
VIEW (1, 1)-(318, 120), , 1
' Creamos un nuevo sistema de coordenadas, en función del nuevo puerto de
' visualización gráfico.
WINDOW (0, 0)-(maxx, maxy)
' Fijamos el área del puerto de visualización de TEXTO
VIEW PRINT 18 TO 22

DO
  PRINT "PRESIONA CUALQUIER TECLA..."

  kolor = kolor + 1
  IF kolor > 3 THEN
    kolor = 0
  END IF

  'Circulo de radio aleatorio
  CIRCLE (160, 100), INT(RND * (maxy / 2)) + 1, kolor
LOOP WHILE INKEY$ = ""

PRINT : PRINT "-- AHORA BORRAREMOS LOS GRAFICOS --"
WHILE INKEY$ = "": WEND
'Borramos los gráficos
CLS 1

PRINT : PRINT "-- AHORA BORRAREMOS EL TEXTO --"
WHILE INKEY$ = "": WEND
'Borramos el texto
CLS 2

WHILE INKEY$ = "": WEND
END
```

Nota: CLS 0 borra toda la pantalla (gráficos y texto), CLS 1 borra únicamente el puerto de visualización de gráficos y CLS 2 borra únicamente el puerto de visualización de texto.

DRAW

La sentencia **DRAW** nos permite dibujar algo conforme a los parámetros que le demos.

Sintaxis Basica:

DRAW cadena_de_comandos

Antes de cualquier cosa, veamos el siguiente ejemplo:

```
SCREEN 1 : DRAW "c2bm100,120d50r50u50f20" :END
```

Lo que significan los comandos de la cadena es lo siguiente:
c2 (Usaremos el **color** 2 de la paleta actual), **bm**100,120 (**ir** a la posición 100, 200 de la pantalla), **d**50 (dibuja 50 pixeles hacia **abajo**), **r**50 (50 pixeles hacia la **derecha**), **u**50 (50 pixeles hacia **arriba**), **f**20 (20 pixeles hacia **abajo derecha**). Ahora vamos a ver una lista con varios comandos:

- Cn: Color n, indica que se va a utilizar el color n.
- Dn: Down n, dibuja n pixeles hacia abajo.
- Un: Up n, dibuja n pixeles hacia arriba.
- Ln: Left n, dibuja n pixeles a la izquierda.
- Rn: Right n, dibuja n pixeles a la derecha.
- En: Dibuja n pixeles en diagonal hacia arriba derecha.
- Fn: Dibuja n pixeles en diagonal hacia abajo derecha.
- Gn: Dibuja n pixeles abajo izquierda.
- Hn: Dibuja n pixeles arriba izquierda.
- Mx,y: Move x,y , se mueve al punto x,y dibujando pixeles (traza una línea desde donde este al punto x,y).
- BM x,y: se mueve al punto x,y sin visualizar.
- Ax: Gira una figura hecha con DRAW, si x=1 entonces la gira 90°, x=2 entonces 180°, x=3 entonces 270°.

Creo, pues, que con esto es suficiente en cuanto a gráficos; con el uso de las sentencias ya vistas, puedes crear una gran variedad de programas según se te ocurra. Ahora pasemos a ver dos funciones que nos permiten emitir sonidos por la bocina interna del ordenador, estas son **BEEP** y **SOUND**.

BEEP

Esta sentencia emite un "beep" por la bocina interna de la computadora.

```
CLS: BEEP : END
```

SOUND

Emite un sonido de una frecuencia y una duración determinada por la bocina interna del ordenador.

Sintaxis:

SOUND frecuencia, duración

- El parametro frecuencia es un entero entre 37 y 32767 que indica la frecuencia (en hertz) del sonido.
- El parámetro duración es un entero de 0 a 65 353 que indica la duración (en ciclos de reloj) que tendrá el sonido. Un segundo es igual a 18.2 ciclos de reloj.

```
CONST maxx = 320, maxy = 200, mitadx = maxx / 2, mitady = maxy / 2
SCREEN 1
DIM pelota(70) AS INTEGER
'Dibujamos la pelota
CIRCLE (mitadx, 10), 10, 1: PAINT (mitadx, 10), 3, 1
```

```
'Copiamos la pelota
GET (mitadx - 10, 0)-(mitadx + 10, 20), pelota
'Borramos la pelota
PUT (mitadx - 10, 0), pelota

'Simulamos la caida
FOR c = 0 TO 179
   PUT (mitadx - 10, c), pelota
   FOR i = 1 TO 10000: NEXT i
   PUT (mitadx - 10, c), pelota
   SOUND 40 + c, 1
NEXT c
```

---

jafet_81@yahoo.com

Capítulo 9 : Introducción a la Programación Orientada a Objetos (POO)

¿QUÉ ES POO?

Para empezar, podemos decir que la POO es una extensión natural de la programación estructurada en la que los procedimientos y datos se *juntan* para formar **objetos**, los cuales se comunican entre sí mediante **mensajes**, para solucionar problemas.

MECANISMOS BÁSICOS DE LA POO

Al parecer, la forma en que trabaja la POO esta basada en *objetos, clases, métodos y mensajes*.

OBJETOS:

Los datos (propiedades) y procedimientos (métodos) están dentro del mismo objeto.

¿Como es esto?. Facil. Por ejemplo, si quisieramos programar un carro en programación estructurada deberiamos crear procedimientos como *AumentarVelocidad*, *DisminuirVelocidad*, *ConsumirGasolina*... pero esto no basta. ¿Por qué?. Sencillamente porque tambien necesitamos datos como el *color*, *numero de puertas*, *placas*, etc... Lo que podriamos hacer, sería crear los procedimientos, y declarar variables para almacenar estos datos... pero esto tampoco bastaría. No podemos SEPARAR el número de puertas de su color, ni de sus placas (**propiedades**), como tampoco de la manera en que consume la gasolina o de la razon de su aumento de velocidad (**métodos**). Todo esto en CONJUNTO forman el carro (**objeto**).

CLASES:

Una **clase** es una colección de objetos del mismo tipo. Todos los objetos de una clase tienen los mismos metodos, aunque puede variar el valor de sus propiedades. Por ejemplo, siguiendo con el ejemplo anterior, podemos tener una clase que se llame *vehiculos de motor*, y dentro de ella varios objetos del mismo tipo, como un Jeep, una Pick Up, un trailer, una Limosina, un Jetta, etc... todos ellos varían en cuanto a sus propiedades (color, forma, etc.), pero todos tienen los mismos métodos (consumen gasolina, aumentan su velocidad, disminuyen su velocidad, etc.).

MENSAJES:

Cuando se ejecuta un programa orientado a objetos, los objetos reciben, interpretan y responden a mensajes de otros objetos. Si a un objeto se le manda un mensaje, este responde ejecutando un método asociado. Por ejemplo, si al carro se le pisa el acelerador (**mensaje**), este responderá con el método *AumentarVelocidad* (método); si se le pisa el freno (mensaje), el carro respondería con el método *DisminuirVelocidad*.

METODOS:

Como ya se mencionó, un método se implementa en una clase de objetos y determina como debe actuar el objeto cuando recibe un mensaje. Un método tambien puede mandar un mensaje a otros objetos solicitando una acción o una información.

CARACTERISTICAS DE LA POO

Al parecer, las caracteristicas principales de la POO son *abstracción, encapsulación, herencia y polimorfismo.*

ABSTRACCIÓN:

Al parecer, mediante la abstracción conseguimos no detenernos en los detalles concretos de las cosas, sino generalizar y centrarse en los aspectos que permitan tener una visión global. Al parecer, una clase es una abstracción, porque los objetos dentro de ella la pueden representar de una manera general.

ENCAPSULAMIENTO:

Al parecer, la encapsulación permite manejar los objetos como unidades básicas, permaneciendo oculta su estructura.

HERENCIA:

Al parecer, la herencia es el mecanismo para compartir automáticamente métodos entre objetos, y crear nuevos objetos a partir de los ya existentes.

POLIMORFISMO:

Al parecer, esta caracteristica permite implementar múltiples formas para un mismo método, dependiendo de la clase sobre la que se realice la implementación. Por ejemplo, podemos aumentar la velocidad de un coche o de una bicicleta, o abrir una lata o una bolsa, etc.

INTRODUCCIÓN A VISUAL BASIC

Visual Basic combina la potencia de los objetos con la sencillez del lenguaje BASIC. Al parecer, Visual Basic soporta la abstracción, el encapsulamiento y el polimorfismo.

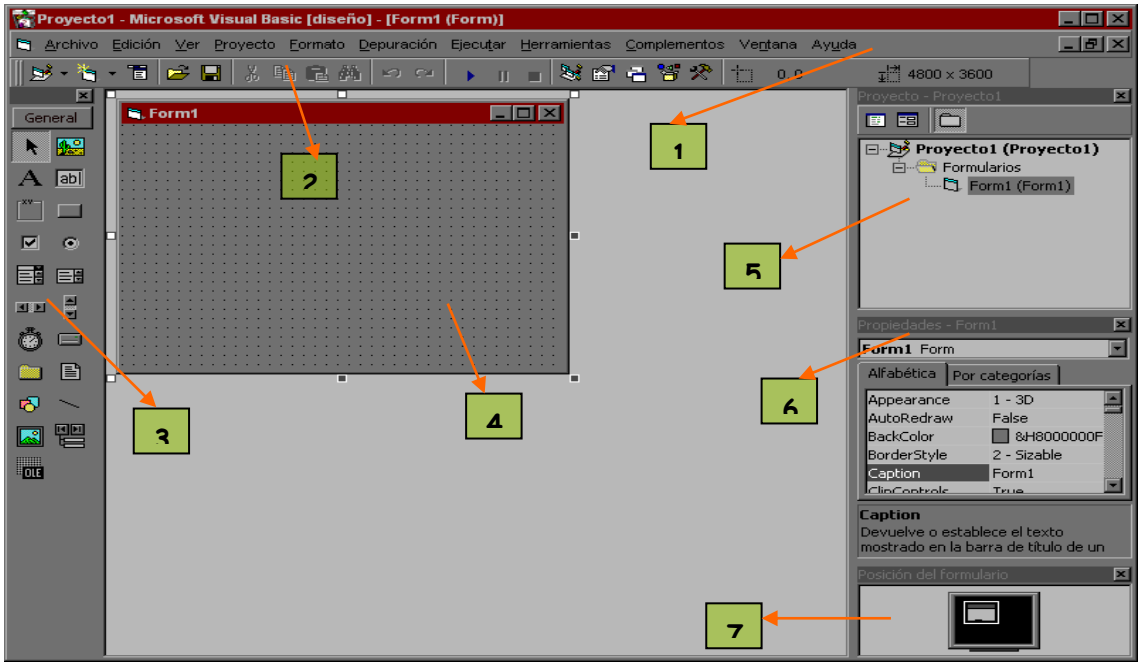
Visual Basic para Windows, nos permite crear aplicaciones para el sistema operativo Microsoft Windows.

PROGRAMACIÓN CON VISUAL BASIC

Para empezar a programar con Visual Basic para Windows, es necesario tener instalada alguna version de Visual Basic para Windows, preferiblemente mayor o igual a la 3.0.

Ahora vamos a dar un paseo por el entorno integrado de desarrollo (IDE, *Integrated Development Enviroment*) de VB.

Nota: El aspecto del IDE puede variar de acuerdo a la version de Visual Basic. El aspecto de la imagen corresponde a la versión 5.0, que es en la que me basaré.

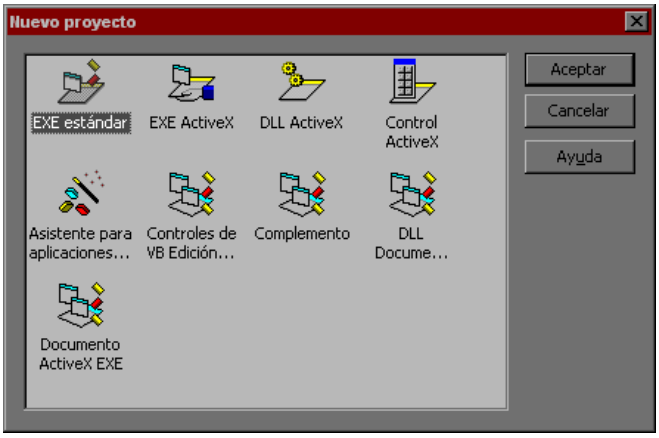


Los principales elementos que conforman el IDE de VB son:

1) **La barra de menus**, en la que encontramos varios menus.
2) **La barra de comandos**, con varios accesos directos a los comandos mas usados como rehacer, deshacer, guardar, compilar, agregar...
3) **El cuadro de herramientas**, en la que estan los diferentes **objetos** que podemos dibujar en el.
4) **El Formulario**, el cual es la base de cualquier proyecto de VB. El formulario es un objeto que contendrá, a su vez, objetos del cuadro de herramientas.
5) **El Explorador de proyectos**, el cual nos permite ver los diversos elementos de que consta el proyecto.
6) **La Ventana de propiedades,** nos permite asignar valores a las propiedades de los objetos que esten seleccionados. Ademas nos muestra la clase a la que pertenece el objeto seleccionado (Form1 pertenece a la clase **Form**), y una breve descripción acerca de la propiedad seleccionada.
7) **Ventana Posición del Formulario**, el cual nos permite manejar la colocación que queramos que tengan los formularios en la pantalla al momento de correr el proyecto.

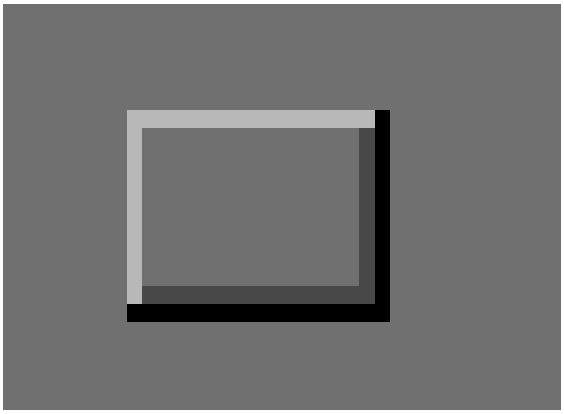
Vamos a crear un ejemplo sencillo: un programa que diga "Bienvenido a Visual Basic".

Para empezar iniciemos Visual Basic. Al iniciar, deberá aparecer una ventana como la siguiente:

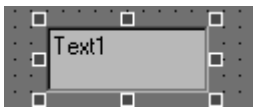


Seleccionemos EXE estándar y demos <ENTER>. A continuación nos deberá aparecer la ventana del IDE con un formulario en el que empezaremos a trabajar. Ahora, demos doble clic sobre el componente TextBox (Caja de texto) del cuadro de herramientas.

Deberá aparecer sobre nuestro formulario el control seleccionado, mas o menos de esta manera:

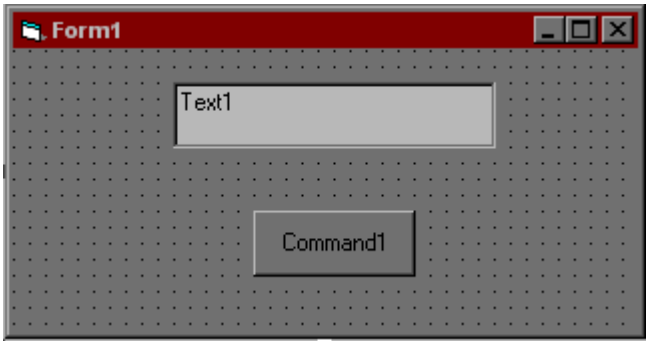


Podemos modificar su posición con el ratón. Si lo seleccionamos deberán aparecer unos cuadritos alrededor de el, de donde lo podemos estirar o encoger.

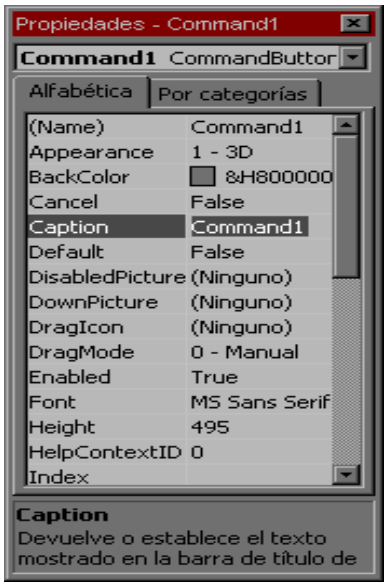


Ahora coloquemos un CommandButton (Boton de comando) de la misma forma.

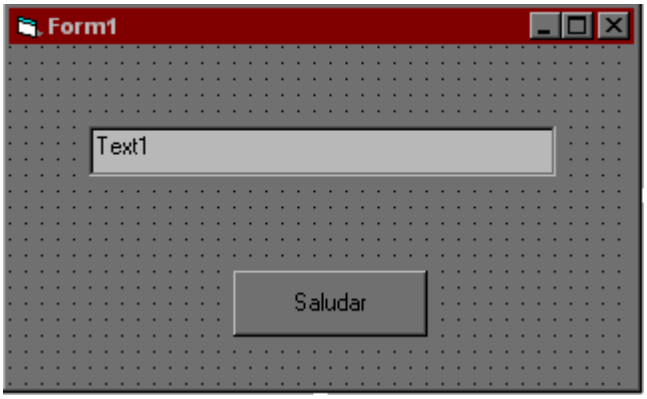
Puedes acomodar los controles para que queden como quieras, así como cambiar el tamaño del formulario y su posición cuando se ejecute el proyecto. Podría quedar algo mas o menos como esto:



Ahora vamos a programar. Los botones, como objetos, tienen varias propiedades, una de ellas es el texto que contienen, en este caso la propiedad que lo establece se llama **Caption**. Vamos a cambiarla seleccionando el botón, y en la ventana de propiedades la buscamos y la cambiamos para que en vez de Command1 diga Saludar.



Ahora el formulario se verá mas o menos así:

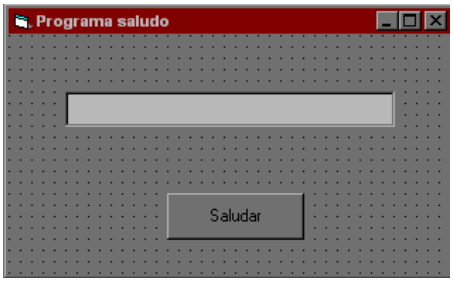


Ahora cambiemos las propiedades de los otros objetos de la siguiente forma:

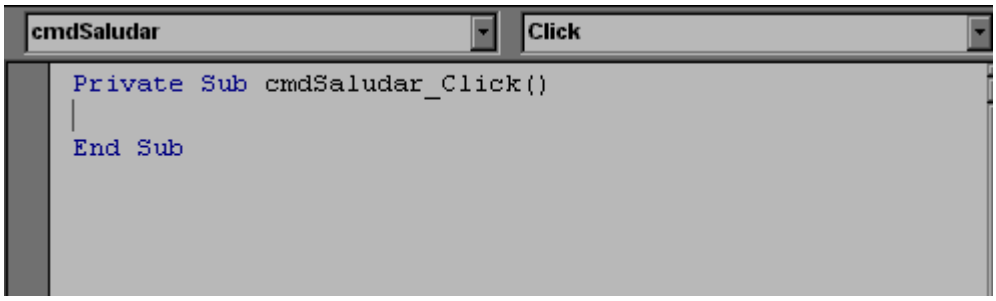
Objeto	Form	CommandButton	TextBox
Propiedades	Name: **frmSaludo** (Nombre con el que será identificado dentro del programa)	Name: **cmdSaludar** (Nombre con el que será identificado dentro del programa)	Name: **txtMensaje** (Nombre con el que será identificado dentro del programa)
	Caption: **Programa saludo** (Es el titulo que tendra el formulario cuando sea ejecutado)	Caption: **Saludar** (Es el texto que tendrá el boton)	Text = **(nada)** (Es el texto que contendrá la caja)

Aquí hay algo interesante que podemos analizar. Si nos fijamos, el nombre de los objetos consta de un **prefijo** de tres letras, mas un **nombre** cualquiera que nos dé una idea de para que se va a usar ese objeto. El prefijo de los nombres, nos da la idea de **que tipo de objeto** es, por ejemplo el nombre del formulario tiene el prefijo *frm*, que nos indica que es un formulario; el nombre del boton de comando tiene el prefijo *cmd*, que nos da a entender que el objeto es un boton de comando, etc. Así existen varios prefijos para los diferentes objetos como *pct* (para un PictureBox o caja de imagen), *lbl* (para una etiqueta o Label), etc. Esta notación permite tener un mayor grado de control sobre los objetos. De todas formas cada quien puede usar la notación que quiera.

O.K, ya con los cambios anteriores, el formulario debería quedar más o menos así



Empezaremos por decirle a *cmdSaludar* (objeto) como va a reaccionar con el evento click. Para eso, demos doble clic en el boton. Nos aparecerá la siguiente pantalla:



Estamos dentro de una SUB como en QuickBASIC, pero ¿que es eso de **Private**?. Esta palabra quiere decirnos que ese **método** solo es exclusivo de ese formulario y no puede usarse fuera de el. Ahora si analizamos el nombre del método, encontraremos que se llama cmdSaludar_Click(). Dentro del cuerpo del **método** iran las instrucciones que le díran al objeto *cmdSaludar* como reaccionar al evento *Click*. O sea el formato es *Objeto_Evento*. Si desplazas la lista de eventos, podras ver todos lo eventos que pueden aplicarse a ese objeto.

Pasemos a escribir las siguientes intrucciones:

```
Private Sub cmdSaludar_Click()
   txtMensaje.Text = "Bienvenido a Visual Basic"
   cmdSaludar.Caption = "Saludar"
End Sub
```

Ahora, vamos a analizar las intrucciones que escribimos en el cuerpo del método.

1) La primera intrucción, **accederá** a la **propiedad** Text del objeto txtMensaje y le da la cadena "Bienvenido a Visual Basic"
2) Las segunda intrucción le dará a la propiedad Caption del objeto cmdSaludar la cadena "Saludar".

Ahora sigamos con la caja de texto. Para ver de nuevo el formulario y los objetos, seleccionamos en el menú Ver el comando Objeto, y de nuevo aparecerá el formulario. Demos doble clic en la caja de texto. Aparecerá lo siguiente:

```
Private Sub txtMensaje_Change()

End Sub
```

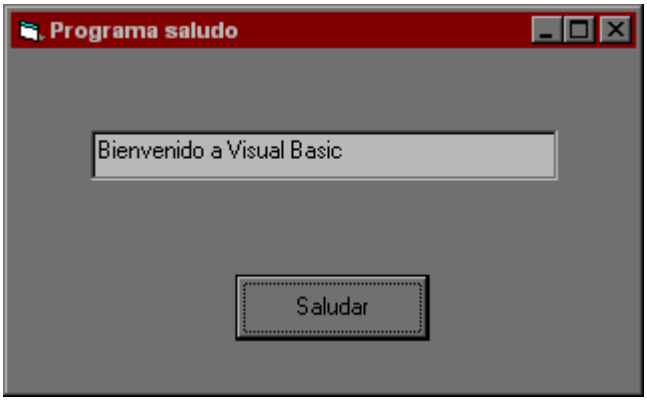
Aquí el evento *Change* se refiere al efecto de modificar el texto de la caja, como borrarlo, agregar otro texto o espacios, etc. La instruccion que se ejecutará cuando suceda este evento deberá ser...

Que significa que la propiedad *Caption* del objeto *cmdSaludar* tendrá la cadena "Restaurar msg".

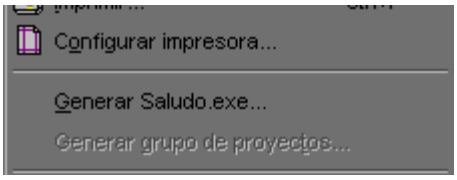
En resumen lo que deberá hacer el programa es:

- Al presionar el boton deberá aparecer el mensaje "Bienvenido a Visual Basic" en la caja de texto.
- Si modificamos el contenido de la caja de texto, el boton dirá "Restaurar msg".
- Si volvemos a presionar el boton, el texto de la caja volverá a mostrar el saludo y el boton tendrá el texto "Saludar"

Ahora vamos a probarlo, dando clic en el botón Iniciar ( ) de la barra de comandos.



Ahora si quieres crear el ejecutable, primero tienes que guardar el proyecto con un nombre (por ejemplo, Saludo) y luego seleccionar Generar Saludo. Exe del menú Archivo.



Espero que haya quedado entendido como se manejan los objetos, clases, métodos, eventos y mensajes. Por mi parte es todo y espero que este curso les haya sido útil. Actualmente puedes encontrar cursos de varios temas relativos a la programación de computadoras en lawebdelprogramador.com.

"Camino a la vida es guardar la instrucción; pero quien desecha la reprensión, yerra." (Proverbios 10:17)

Curso de QuickBASIC 5ª Ed., por Tadeo E. Ortega Ch.
jafet_81@yahoo.com